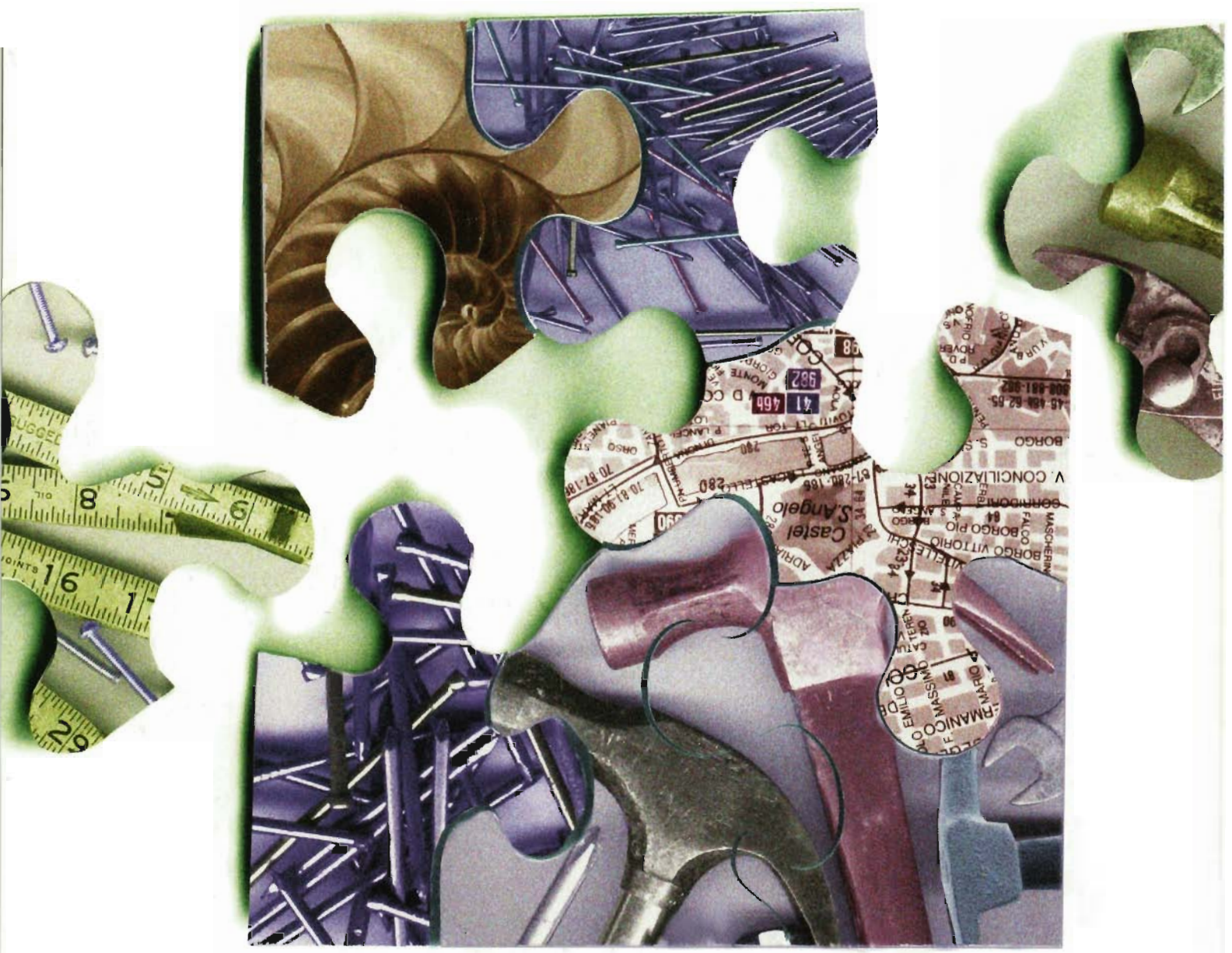
GRAPHICAL SOFTWARE DEVELOPMENT

SYSTEMS ENGINEERING

Digital Technical Journal



Volume 7 Number 2
1995

Cover Design
DEC FUSE software, described in this issue, is an integrated set of programming tools for UNIX developers. Our cover image of unique puzzle pieces being fitted together conveys the DEC FUSE concept of an integrated collection of independent tools cooperating within a graphical development environment.

The cover was designed by Lucinda O'Neill of Digital's Design Group.

The *Digital Technical Journal* is a refereed journal published quarterly by Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460. Subscriptions to the *Journal* are $40.00 (non-U.S. $60) for four issues and $75.00 (non-U.S. $115) for eight issues and must be prepaid in U.S. funds. University and college professors and Ph.D. students in the electrical engineering and computer science fields receive complimentary subscriptions upon request. Orders, inquiries, and address changes should be sent to the *Digital Technical Journal* at the published-by address. Inquiries can also be sent electronically to dtj@digital.com. Single copies and back issues are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are also available on the Internet at http://www.digital.com/info/DTJ/home.html. Complete Digital Internet listings can be obtained by sending an electronic mail message to info@digital.com.

Digital employees may order subscriptions through Readers Choice by entering VTX PROFILE at the system prompt.

Comments on the content of any paper are welcomed and may be sent to the managing editor at the published-by or network address.

ISSN 0898-901X

Documentation Number EY-U001E-TJ

Book production was done by Quantic Communications, Inc.

Contents

Editor's Introduction

The integration of distinct parts to form a useful and effective whole is the underlying theme for two sets of topics in this issue. The opening papers describe the integration of programming tools to create a graphical software development environment. The second set of papers addresses the integration of large, complex systems—systems that encompass all the software and hardware components needed to serve the user's purpose.

The DEC FUSE software development product is designed to take advantage of UNIX workstations' graphical capabilities, supporting such programming languages as C, C++, and Fortran. Rich Hart and Glenn Lupton review the origins of DEC FUSE in the FIELD environment developed at Brown University and compare FUSE with similar environments based on a tool integration model. The authors present two key aspects of the product design: graphical user interfaces built on top of UNIX commands and a multicast messaging mechanism that allows the tools to work together.

A tool recently integrated into the DEC FUSE suite is the Data Visualizer, which allows software developers to display thousands of lines of code with associated statistics. Don Zaremba describes the process of taking the tool from advanced development through implementation and relates what the engineers learned as they adapted current visualization research to their goals and built prototypes of the technology. He concludes with a description of the resulting product and plans for future work.

Our next three papers explore experiences with different aspects of systems-level engineering and integration. Eric Newcomer's overview of the Multivendor Integration Architecture (MIA) effort, initiated by Nippon Telegraph and Telephone (NTT), highlights many factors that in general make systems integration challenging. NTT sought, through standardization, to resolve the costly problem of incompatible application environments. Eric discusses the MIA's chosen direction based on the need for portability, interoperability, and a common user interface. He then describes Digital's contribution in the area of distributed transaction processing and summarizes the MIA consortium's successes and continuing work.

A specific object-oriented product developed to integrate systems applications is the subject of Jim Kirkley's and Wick Nichols' paper. Comprising Jacobson's and Rumbaugh's methodologies, third-party software, and Digital's CORBA-compliant ObjectBroker, the Framework-based Environment (FBE) product addresses the need for new and legacy applications to interoperate in a distributed manufacturing system. The authors step through a typical integration project and expand on trade-offs that must be addressed in an integration project that takes an object view of the system environment.

A major systems engineering project to solve the problem of ongoing introductions of software into a large computer network is described in the concluding paper by Owen Tallman. The project, commissioned by a large French bank, extended over a network of data center clustered servers, branch servers, and thousands of workstations and personal computers. Owen outlines the customer's requirements and Digital's role as developer of the automated software deployment facility. He reviews the configuration management model (CMM) and other models that were the basis for the project team's work. His discussion of the implementation encompasses examples that illustrate the intricacies of a rigorously managed software deployment process.

The editors thank Mikael Rolfhamre of Digital's UNIX Business Segment, Ed Balkovich of Digital's Corporate Research Group, and Hank Jakiela of the Systems Business Unit for their help in developing this issue. At the end of the issue, we also acknowledge and thank the referees for their very valuable reviews of manuscripts submitted during this past year.

Upcoming topics in the Journal are Digital's high-performance Fortran compiler and parallel software environment, and the Sequoia 2000 global change research project.

Jane C. Blake
Managing Editor

Foreword

Mahendra R. Patel
Corporate Consulting Engineer
Vice President, Systems Engineering

Systems engineering is the engineering of complete systems as opposed to parts of systems. Exactly what this means depends on one's point of view. One person's system is another person's component. From chips to boards to boxes to clusters to networks, subsystems are combined into ever larger and more complex aggregates. At Digital, systems engineering means the engineering of systems at a level of aggregation above individual hardware or software products. Individual processors, storage subsystems, network hubs, operating systems, database systems, and applications are viewed as components of the system. For example, a nationwide network for interactive securities trading, built from hundreds of nodes at dozens of sites, is one system.

A number of trends in the computer industry make it more challenging for a computer company to practice systems engineering:

- Commoditization: Component products, from microprocessors to applications, are increasingly becoming low-cost, high-volume commodities. Ironically, as the cost of the components drops, the cost of integrating them into complete systems becomes a larger fraction of total system cost.
- Distributed systems: While they provide new opportunities for better performance, scaling, and fault-tolerance, distributed systems also present new engineering challenges for ensuring these same attributes.
- Heterogeneous systems: Increasingly, computers from a variety of vendors, running a variety of operating systems, are being connected together and are expected to work together correctly.
- Complexity: Distributed systems are becoming more complex for a number of reasons. The number of components is growing. The number of types of components that must work together is growing. And the variety of unique configurations is growing.

During the last decade, the computer industry has changed from one that offered vertically integrated systems built from proprietary CPUs, disks, networks, operating systems, and layered products to one that produces commodity products conforming to de jure or de facto standards. Unlike the manufacture of automobiles or aircraft, a single computer manufacturer seldom produces all the components of a complete working system. The hardware, system software, and applications often come from three different vendors. Systems engineering, as now practiced in the computer industry, places less emphasis on top-down design of hardware and software components and their interfaces to meet system-level goals. Rather, it is based on anticipating a broad spectrum of system designs.

From the point of view of a computer company, systems engineering must now be concerned with assemblies of commodity hardware and software products. Thus, four areas are of special interest to systems engineering in the computer industry: interoperability, performance, scalability, and availability.

Interoperability of components, including components from different vendors, is difficult to verify because

of the virtually infinite number of possible combinations of components. For example, the introduction of a new component often can expose bugs in system components previously thought to be working. Systems engineering work in this area includes the development of tools for effective testing and the development of industry standards for interoperability.

The performance of a system can depend in a complicated way on the performance of its components. Sophisticated tools are needed to predict the performance of a complex system from the performance of its parts or to diagnose subtle interactions between components. Today, performance tools for distributed systems are not as sophisticated as those for individual computers.

Scalability refers to the ability of a system to start small and grow big. Size may be measured in terms of numbers of users, computers, disks, applications, or a combination of parameters. The ability to scale up distributed systems over two orders of magnitude by adding components is one of their most attractive attributes. However, scaling effectively requires careful analysis and design of the system. For example, a system design based on cost-effective packaging of functionality at a small scale can exhibit bottlenecks as computers are added to the system to handle increased workloads.

A distributed system is inherently less reliable unless care is taken to improve availability by adding redundant components. Simply partitioning functionality between a client and server computer requires that both the client *and* the server be working for the functionality to be available. Given technology with the same failure and repair characteristics, distributing functionality between two computers results in a system that is less available than one with the complete functionality on one computer. Often this is an academic point in simple systems, given the levels of component reliability. However, distributed systems with critical availability requirements (e.g., a nationwide network for interactive securities trading) demand careful analysis and design to add appropriate redundancy.

Systems engineering is important to Digital because even the best component products are of no value to customers until they are integrated into complete working systems that meet business needs. Ideally, one would like to be able to build large, complex systems by simply snapping together small, simple components, as if they were Lego blocks. It is tempting to assume that this should be easy because many of the components are available as inexpensive, mass-produced, reliable commodities. However, building complex systems from simple parts is still difficult and requires engineering work, especially when the overall system stretches the limits of the technology.

Systems engineers play a vital role in major systems integration projects that push the edge of the technology envelope in some way. The system may combine components never before used together. The trend toward heterogeneous systems makes this more likely. The system may stretch scaling limits by having more nodes or network connections or users or data than ever before. The trend toward large distributed systems makes this scaling possible. The system may need to meet demanding requirements for overall system performance or dependability. Increasingly, heterogeneous, distributed systems are being used for mission-critical business applications.

Engineering analysis and design is needed at all phases of a complex integration project, from the definition of the technical requirements to the design of the system to final testing and verification. Custom software or hardware may need to be developed, either to glue together components that were not built to work together or to substitute for standard components in order to meet demanding requirements for performance or scaling.

Systems engineers also develop tools and methods to simplify the task of integrating complete systems. Digital's systems engineers are active in the development of industry standards for ensuring the interoperability of components from different vendors. In this issue of the *Journal,* Eric Newcomer's paper describes the development of standards for use in the telecommunications industry. Often, a system has legacy components. Digital's systems engineers are also active in the development of frameworks that apply object-oriented programming technologies to encapsulate legacy applications and data, simplifying the incorporation of legacy components into new systems. A framework for the integration of manufacturing applications is described in the paper by James Kirkley and William Nichols. The Systems Engineering group has developed test tools and methods, and operates an extensive laboratory for testing, verification, and performance characterization of combinations of products from Digital and other vendors. Testing and characterization data are the basis for configuration guidelines for systems intended to run a number of popular commercial applications.

Computers, disks, network switches, database systems, desktop applications, and many other components are now available as inexpensive, reliable commodities. Hardware and software components from various manufacturers can be put together to build a wide variety of systems, from one as simple as a PC to one as complex as a worldwide distributed system.

While the cost of the components has dropped dramatically in recent years, the cost of integrating these simple components into complex distributed systems remains high and therefore represents a larger fraction of the total cost of the system. Today, Digital's ability to successfully build complex distributed systems provides great value for our customers, often greater than the value of the commodity components from which the systems are built. For the future, improvements in tools and methods for building complex systems will lower the cost of these systems significantly, making new types of applications feasible and affordable.

Richard O. Hart
Glenn Lupton

DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools

DEC FUSE is an integrated programming environment for UNIX systems. It is an evolution of the FIELD environment developed at Brown University. To take advantage of the features of workstations developed during the 1980s, these environments were designed to provide graphical user interfaces for commands commonly used by UNIX software developers. DEC FUSE uses two methods to create an environment from smaller and simpler software components. These methods are sending messages between components and layering graphical interfaces on top of UNIX commands. DEC FUSE uses these methods to create an easy-to-use, integrated environment with more features than its individual components.

The UNIX operating system originated at Bell Laboratories in 1969 and rapidly grew more popular, first within Bell Labs, then at universities and, since the early 1980s, at commercial enterprises. One reason cited for its success is that it is a good operating system for programmers.[1] The wealth of simple tools and the ability to combine them easily into new tools provides an attractive environment for software development. Projects organize their development processes around the capabilities of UNIX tools like sccs for version control and make for application building. Developers build project-specific tools using UNIX commands in shell scripts and have become proficient in the use of tools like the dbx debugger and the emacs and vi editors.[2] Developers have also become accustomed to commands for text manipulation (sed, awk), searching (grep), and comparing (diff), and the use of these in combination with other commands to do special tasks.

In the late 1980s, workstations came into common use for software development. Workstations provided additional compute power and were capable of displaying complex graphics and providing point-and-click interfaces. The UNIX tools and shell environment, designed around character-cell video terminals and hard-copy devices, did not make effective use of these workstation capabilities. Different tools and a different approach to combining them were needed to provide an effective workstation-based development environment that would take advantage of the additional compute power available to workstation users and the graphical interfaces available using the X Window System.[3]

In this paper, we define the characteristics of some integrated software development environments designed to take advantage of modern UNIX workstations. We describe the DEC FUSE product as an example of one of these environments and present two methods used to create the DEC FUSE product. With the first method, we show how tools are built as graphical user interfaces (GUIs) on top of existing UNIX commands. Then, we show how messaging enables these tools to work together. We present trade-offs and design alternatives for each method.

Integrated Software Development Environments

Integrated software development environments are collections of software programs, or tools, that are used together to accomplish one or more phases of software development. DEC FUSE and other integrated software development environments, including HP SoftBench from Hewlett-Packard and SPARCworks from Sun Microsystems, are based on a control integration model.[4–7] Control integration enables tools to make requests of other tools for information or to do required tasks.[8]

The DEC FUSE, HP SoftBench, and SPARCworks environments were strongly influenced by work done at Brown University on the FIELD programming environment by Steven P. Reiss.[8,9] DEC FUSE, in fact, continues to use some code originally written as part of FIELD. These environments share the following features with FIELD:

- Environments are collections of cooperating tools. Each tool addresses a single aspect of the software development process such as editing, searching, debugging, or building. This follows the UNIX philosophy of making tools or commands simple and focused on a single problem. As a result, they are easier to build, maintain, and use. The tools cooperate with each other by performing operations at the request of other tools. For example, the builder tool can request that the source code corresponding to an error be displayed, and the text editor will present the code.
- Tools use a selective broadcasting communications method. Tools send simple, usually textual, messages to communicate with other tools.[10] A message may be either a request for a service or a notification of the occurrence of an event. Tools register their interest in receiving particular messages. A message is then broadcast without requiring the sender to specify who will receive it. Since requests are not directed to a particular tool, a tool can be replaced with a similar tool that responds to the same messages without making changes to the sender. Because messages are broadcast, multiple tools can receive a notification and each can take appropriate action.
- Source files and annotations are viewed using a single text editor. Each tool that needs to present source text to the user does so by sending request messages that are processed by a single source text editor. The text editor displays the desired source files, and it may also place annotations next to source lines of interest. Annotations are used to link the sources with other parts of the environment. For example, the location of breakpoints is provided by the debugger, the location of build errors by the builder, and the location of strings matching a pattern by the search tool. Each of these locations is identified with an annotation symbol next to a line of source code in the editor display.
- GUIs are built on top of UNIX tools. Many of the tools in the environment are GUIs fitted to existing UNIX commands such as make, grep, and dbx. These interfaces provide menu and button access to these commands and their options; they also interpret the results of the commands, presenting them in formatted, interactive displays.
- Program information is presented pictorially. The graphical display capabilities of the workstation are used to pictorially present information that may be complex or extensive. For DEC FUSE, this includes a program's function call graph, the dependencies in a makefile, or the execution times of each function in a program. This issue of the *Digital Technical Journal* presents another example of displaying information pictorially with DEC FUSE in the paper "Adding a Data Visualization Tool to DEC FUSE."[11]
- Users continue to use familiar tools and methods. Because the FIELD and DEC FUSE environments are built using existing tools such as make, sccs, and dbx, users can continue to use tools with which they are familiar. They can also use existing makefiles and source libraries in the environment. In addition, users can make a gradual switch to an environment such as DEC FUSE. They can use DEC FUSE when it is most advantageous and continue to use older tools and methods when that is preferable.

DEC FUSE Overview

The primary goal of the DEC FUSE product was to create a commercially useful, integrated software development environment supporting a variety of programming languages, including C, C++, and Fortran. The DEC FUSE environment takes advantage of the capabilities of the UNIX workstation, while allowing software developers to preserve their investment in familiar UNIX tools. DEC FUSE designers adopted some FIELD components, which were converted to use Motif. Extensions were also made to the FIELD environment to create the DEC FUSE product. These extensions are described in the next sections. Several tools have been added to the environment through successive releases of DEC FUSE. The tools supplied with DEC FUSE version 2.1 are listed in Table 1 and are described in subsequent sections.

Selective Broadcasting Mechanism

The messaging used by DEC FUSE, called the multicast messaging system, has been extended in two ways beyond its FIELD origins. First, messages have been made more functional in nature. In the FIELD environment, messages are strings that are assembled by

Table 1
Tools Supplied with Digital's DEC FUSE Version 2.1

DEC FUSE Tool	UNIX Commands Used
Editors	emacs, vi (and a Motif-based editor)
Debugger	dbx or DECladebug (on Digital platforms)
Search	grep, fgrep, egrep
Builder	make, gnumake
Code manager	sccs, rcs
Man page browser	man
Cross-referencer Call graph browser C++ class browser	Use common data from compilers or other source scanners.
Profiler	prof, gprof, pixie
Compare	diff
Help	HyperHelp
DEC FUSE shell	sh, csh, ksh, ...

the sending tool and delivered to receiving tools. The receiving tools have registered an interest in particular messages by describing them using a pattern string. DEC FUSE uses a more functional interface that more closely resembles a remote procedure calling mechanism. Each tool defines the messages that it can send and receive as function definitions using the DEC FUSE tool integration language (TIL). Second, a set of components called the DEC FUSE EnCASE facility has been developed to support the integration of new tools and new messages into the DEC FUSE environment.[5] These components include the TIL compiler and the Message Monitor tool, described later in this paper.

Choice of Source Code Editor

Instead of having a different editor as part of each tool, the FIELD environment provided a single GUI-based editor. Because most users have strong preferences about which text editor they use, DEC FUSE extended the environment to allow each user to choose from three different editors: emacs, vi, and the DEC FUSE editor.[2] Both emacs and the DEC FUSE editor support use of annotations supplied through interactions with other tools. Users of the vi editor do not see annotations, but other tools can still position vi on source lines of interest.

DEC FUSE Tools

The tools described in this section are currently available in DEC FUSE. Figure 1 shows the DEC FUSE C++ class browser, builder, code manager, and profiler tools.

- The search tool searches files for strings matching a literal string or regular expression using grep. Options available through the user interface allow for specifying whether the search should be case-sensitive, whether lines matching or not matching should be displayed, and whether the search should be limited to a single directory or an entire directory tree.
- The builder builds applications using the make or gnumake commands and existing makefiles or makefiles generated by the builder. A scrollable results window shows the output for the build operation, including diagnostic messages. The build dependencies between the files for the application that are described in the makefile are displayed graphically. The builder also distributes build actions across hosts on a local area network (LAN) and provides a user interface for specifying those hosts and for monitoring the progress of the build.
- The debugger provides a GUI to command line debuggers. This interface provides a source display with annotations for breaks, conditional breaks, and the current execution point. Debugging commands can be executed using buttons, menus, and a command line interface. Special windows provide for viewing and changing variables, breakpoints, and machine registers, and for monitoring the values of expressions.
- The compare tool displays the differences between two text files in a side-by-side display with related areas highlighted and graphically connected. The analysis of the differences is provided by the diff command.
- The code manager provides a GUI to the version management tools rcs and sccs. The code manager displays the revision history of the managed files. Details such as author, date created, and comment can be displayed for each version. In addition, the code manager uses the compare tool to display differences between revisions or revisions and files.
- The man page browser displays the reference pages for commands, system calls, subroutines, and special files. References to other manual pages in the text are hot links, and the user can click on a reference to display the other page. The man page browser can also display an index of selected reference pages. Users control the index content by specifying a keyword to match in the reference page description or a prefix to match to the reference page name. These allow users to find reference pages when they are unsure of the function or command name.
- The profiler runs an application to collect run-time statistics and displays the results at the function and line level. Statistics include the CPU time used by functions or source lines, function-call counts, line-execution counts, and function and line test coverage.
- The cross-referencer displays source locations for declarations, references, and function calls whose

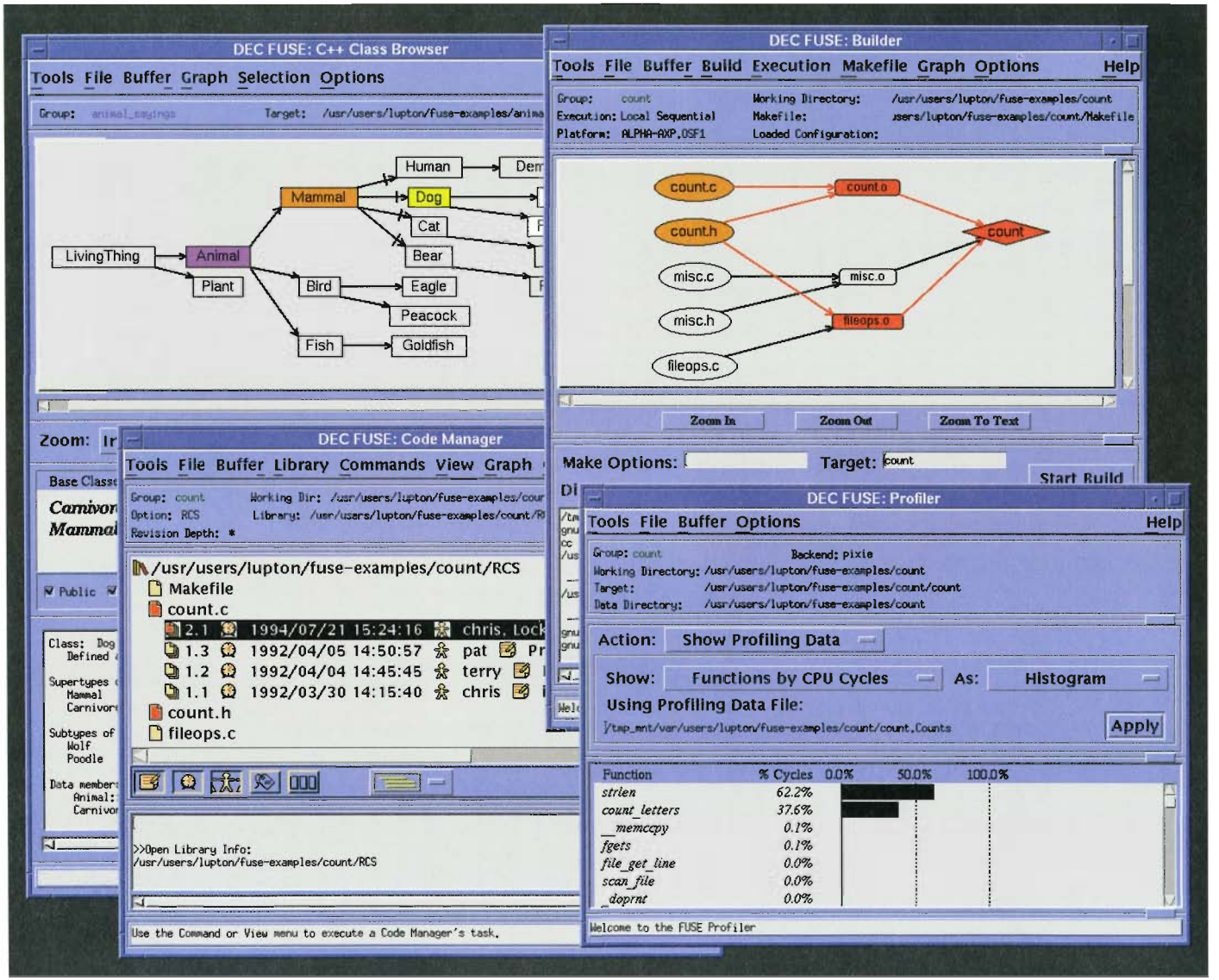


Figure 1
DEC FUSE C++ Class Browser, Builder, Code Manager, and Profiler

names match a regular expression. Queries can be constrained by declaration types and locations among other things.

- The call graph browser graphically displays the call relationships within a program. Relationships between functions, source files, and source directories can be shown. The user can constrain the display to selected parts of the program.
- The C++ class browser displays the C++-class hierarchy graphically. Inheritance paths and detailed information about each member and class can be displayed.
- Editors include the DEC FUSE text editor, emacs, and vi. The DEC FUSE and emacs editors allow other DEC FUSE tools to supply annotations on source text lines of interest. In addition, other DEC FUSE tools can be invoked from the editor, including the builder, the code manager, and the man page browser. The DEC FUSE emacs editor is a standard emacs, with additional keys defined for DEC FUSE functions.
- The help tool works with the HyperHelp tool from Bristol Technology, Inc. to display on-line help and training.
- The DEC FUSE shell supplies a terminal emulator window running a standard UNIX shell in the context of the user's DEC FUSE development environment.

In addition to the tools listed above, DEC FUSE includes a control panel tool that starts tools and manages their environment.

Using the DEC FUSE Tools Together

The messaging mechanism allows each of the tools to make selected operations available to other tools. For example, the editor makes its ability to open and display a source file and to position to a specific line available to the other DEC FUSE tools through messages. The man page browser accepts a message that causes it to display a manual page for a specified topic. The following scenario, summarized in Figure 2, shows how messaging ties together DEC FUSE tools into an integrated environment.

1. To locate places in an application that need to be changed, the developer starts the DEC FUSE search tool and looks through C source files for occurrences of a particular name. The files and lines containing a match are displayed in the search tool. By double-clicking on a line, the corresponding file is loaded into the DEC FUSE editor, and the line is displayed with an annotation that the search tool provided the location. (The search tool is used in this scenario, but the cross-referencer can also be used to do this task.)

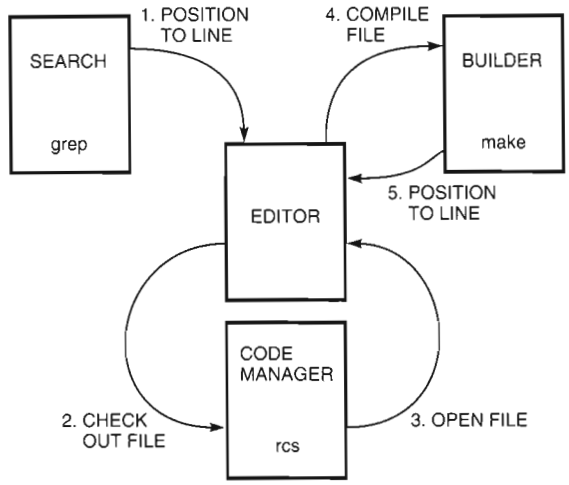


Figure 2
DEC FUSE Tool Communications

2. After inspecting the source, the user decides to modify the code, but must first check it out using rcs. By choosing the "check out" menu item in the editor, the user starts the DEC FUSE code manager, which shows the user the revision being checked out and allows the user to browse the library before confirming the check-out operation.
3. The code manager sends a message to the editor telling it to load the file to ensure that the user is editing the latest version.
4. The user edits the file and then starts a compilation using the "compile file" menu item in the editor. This starts the DEC FUSE builder, which runs make and displays compiler diagnostics.
5. By double-clicking on a diagnostic, the user gets back into the editor on the line containing the error.

The messaging mechanism allows for automated switching between the tools. Information is passed between the tools, thus eliminating retyping or cutting and pasting. Other features also contribute to the feeling of an integrated environment in DEC FUSE. These include consistent GUIs for all tools, global preference setting, saving and restoring of state information, and centralized help and training. However, it is the messaging that ties tools together, making DEC FUSE an integrated environment rather than a simple collection of tools.

We have now examined the features of integrated software development environments in general and the DEC FUSE environment as an example of these environments. In the next two sections, we examine two important aspects of the design of DEC FUSE. First, we discuss the mechanisms used to add graphical interfaces to existing UNIX commands. Then we present the design of DEC FUSE messaging.

Building Graphical Interfaces for Existing UNIX Commands

Most DEC FUSE tools consist of a graphical program that provides a point-and-click interface for invoking UNIX commands. This program interprets the results from the execution of the commands and presents these results graphically. This approach has several advantages over building a completely new tool. These are examined in this section, along with the implementation techniques used.

Rationale for Building a Graphical Interface for Existing Commands

Using an existing command to perform functions needed by a new command is a technique that is often used on UNIX systems. DEC FUSE tools use existing commands for the following reasons:

User Investment Protection Two types of investments must be made in software development environments. One investment is training: software developers have learned the concepts and capabilities of the underlying tools. Since the graphical interfaces of an integrated environment are built on tools that are familiar to users, they can be learned in considerably less time. For example, the concept of revisions, the semantics of revision numbers, and the capabilities of rcs are the same whether rcs is invoked from the command line or selected from the DEC FUSE code manager.

Second, a project may have invested in procedures and software that depend on project tools such as make and sccs. Users often use many makefiles that have been tailored to meet the needs of their project. Likewise, most projects use sccs and rcs in ways that must be supported by scripts. By building the code manager and builder on the existing rcs, sccs, and make utilities, this investment is preserved. (The DEC FUSE code manager provides mechanisms to support user-written scripts used in combination with sccs and rcs.)

Easier to Invoke Operations Although the UNIX command line environment is extremely flexible, most users find themselves frequently referring to reference pages to check command syntax and option flags. By replacing commands with menu items and buttons and by replacing flags with toggle buttons and fill-in-the-blank dialog boxes, users interact with the tools faster with less typing and less browsing through reference pages. This is especially true for novices who have not defined their own collection of aliases and scripts.

For example, searching all the header files in a directory hierarchy for the occurrence of a string requires a command like the following:

```
find /usr/include -name "*.h"
  -exec grep -i FLT_M {} /dev/null \;
```

This is a typical example of a command that a software developer might need to use from time to time. The command would be entered on one line. A first-time user, however, might not correctly input all the details of the command for the following reasons:

- The "*.h" designation includes quotation marks so that it is not immediately expanded by the shell in the user's current directory, but instead expanded by find in all the subdirectories in the /usr/include tree.
- If the search is to be case-insensitive, the -i switch must be used with the grep command.
- The grep command supplies the name of the file where the string is found only if more than one file is supplied in the grep argument list. /dev/null is added to make grep include the file names in the output.
- The find command requires that subcommands that it will execute be terminated with a semicolon. Because a semicolon is also recognized by the shell, it must be preceded with a backslash (escaped), so that find will see it.

To do the same operation from the DEC FUSE search tool, the user fills in some fields and sets a toggle (see Figure 3). This can be done easily and correctly the first time by both novice and experienced UNIX users.

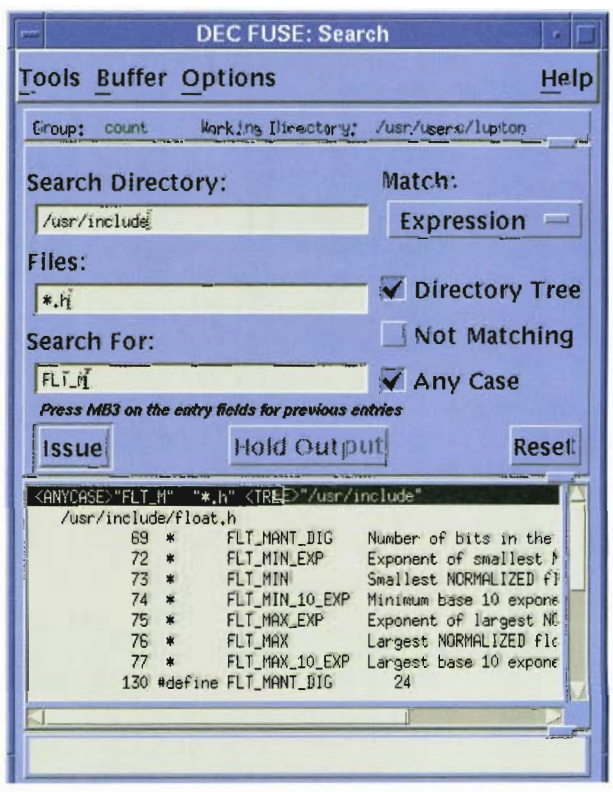


Figure 3
DEC FUSE Search Tool

When the user spots an interesting occurrence in the output from a grep command and wants to edit the file, a command line interface requires the user to enter the command to edit the file and to type the file name and line number. Using the DEC FUSE search interface, the user double-clicks on the interesting line in the search tool and the editor automatically loads the file and sets the position to the desired line, saving typing and eliminating the possibility of errors.

Hiding Details Another advantage of graphical interfaces on underlying commands is the ability to hide details of particular commands. For example, the DEC FUSE code manager supports both sccs and rcs with the same graphical interface. A user does not need to know the differences between rcs and sccs; by using the graphical interfaces, the user can see similar version history information from either underlying library format.

Graphical Presentation One advantage of a workstation is its ability to present information graphically. A GUI layered on a command line tool can analyze the output of the tool and present it to the user graphically, making the information in the output easier to understand.

An example of this is the dependency graph in the DEC FUSE builder, as shown in Figure 4. The graph displays the build dependencies for the user's application as specified explicitly or implicitly in the application's makefile. This display is an analysis and presentation of the output provided by make when run with options that produce debugging information about makefiles. Nodes designated orange in the graph represent the files that have changed. Nodes designated red in the graph represent the files that need to be rebuilt because of their dependency on the changed files.

Another example of using the graphical capabilities of the workstation is the DEC FUSE compare tool, which is built on the UNIX diff utility. The output of the UNIX diff utility is textual; an example is shown in Figure 5. In contrast, Figure 6 shows how the DEC

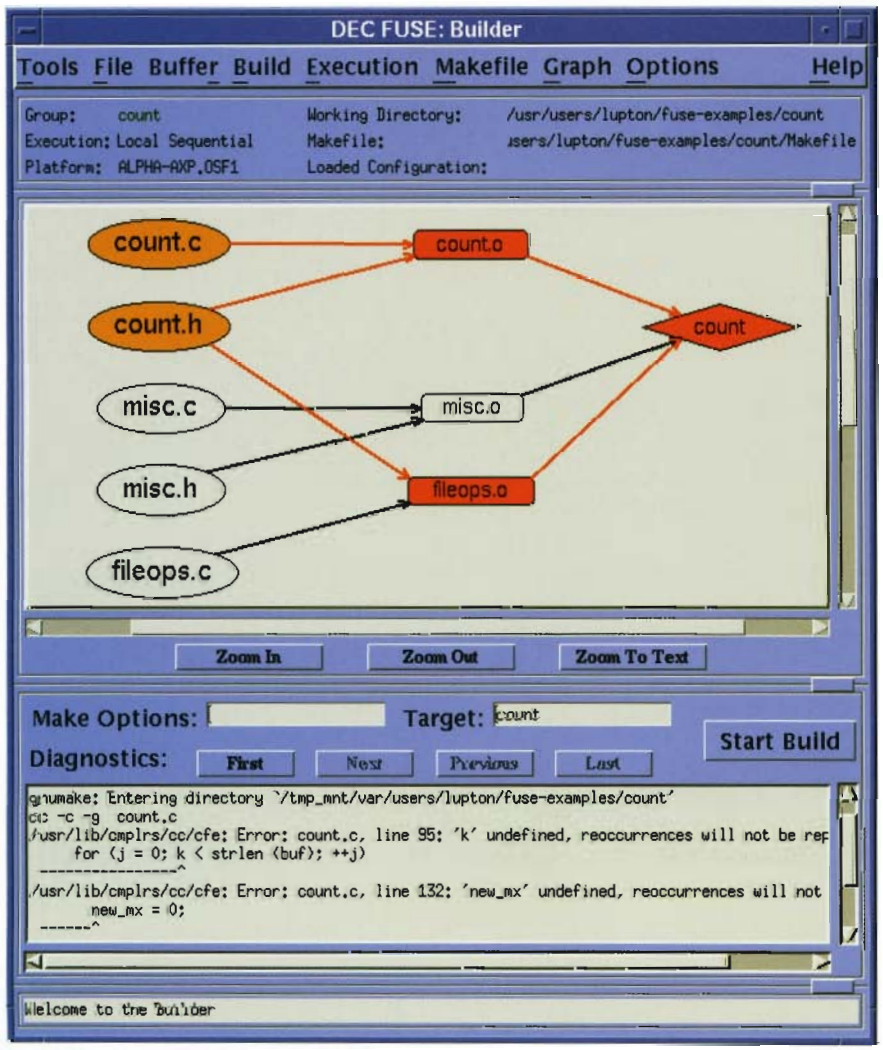


Figure 4
DEC FUSE Builder Tool with Dependency Graph

```
csh# diff file1.txt file2.txt
5,9d4
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
< These are lines that are only in file1.
11a7,10
> These are lines that are only in file2.
> These are lines that are only in file2.
> These are lines that are only in file2.
> These are lines that are only in file2.
14,17c13,16
< These are lines that are different in file1.
< These are lines that are different in file1.
< These are lines that are different in file1.
< These are lines that are different in file1.
---
> These are lines that are different in file2.
> These are lines that are different in file2.
> These are lines that are different in file2.
> These are lines that are different in file2.
```

Figure 5
Sample diff Output

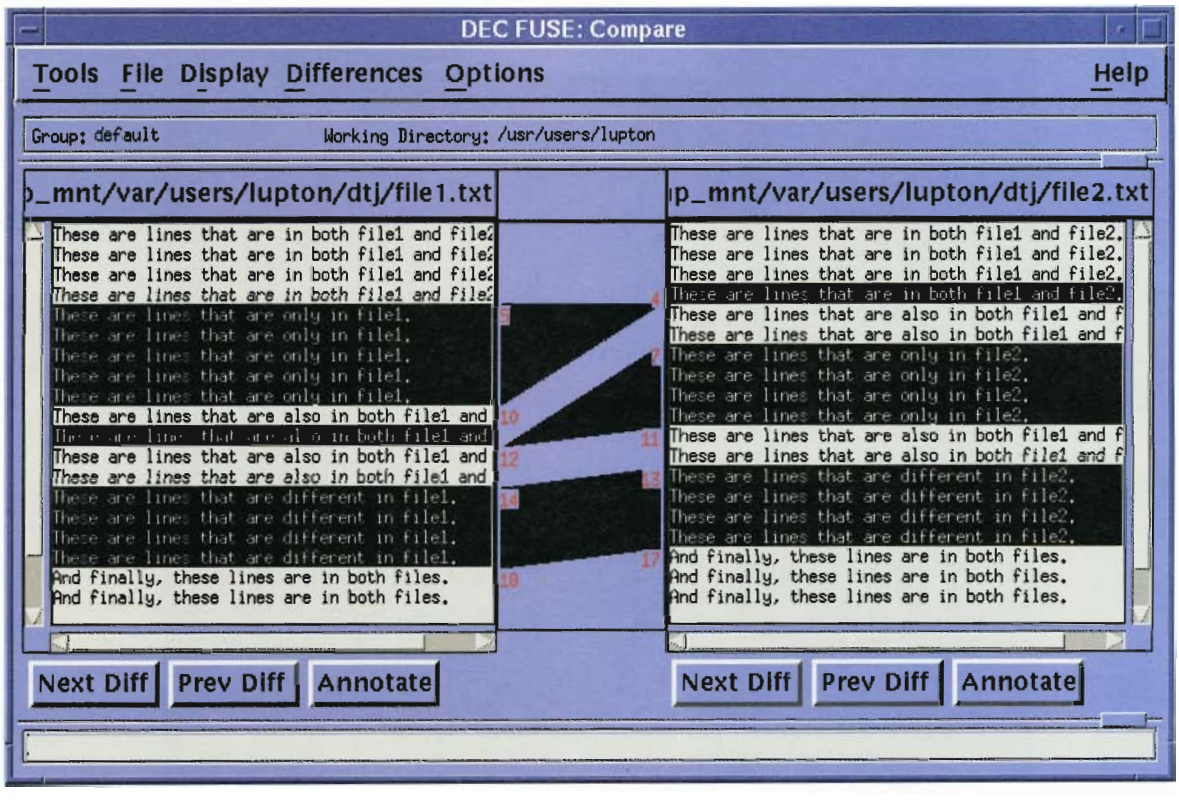


Figure 6
DEC FUSE Compare Tool

FUSE compare utility displays these differences graphically, using highlighting to indicate the differences and shapes to connect regions in the two files that relate. The display allows differences to be viewed in the context of the lines before and after them and the lines that correspond to them in the other file.

Reduced Tool Development Work An obvious advantage for the developers of the interface is that building on a command line tool may involve considerably less work than designing and implementing a new tool that includes all the capabilities of the command line tool. Furthermore, not every capability needs to be provided through the user interface of the tool, because users have access to less-used capabilities through the command line. For example, the seldom-used administrative features of sccs and rcs can be omitted from the user interface. Thus, with a minimum amount of effort, it is possible to provide a convenient interface to the most important underlying capabilities.

Managing Command Interfaces

It is common on UNIX systems to use the output of one tool as input to another. In the case of DEC FUSE, the output of command-line tools is being used as input to DEC FUSE tools. The DEC FUSE tools construct commands and pass them to a separate process for execution. The results of these commands are then interpreted by the DEC FUSE tools so that desired information can be presented to the user. The methods used to issue commands and to analyze their results vary from one DEC FUSE tool to another.

One method used by DEC FUSE tools is to directly issue commands using the popen library function, which both starts execution of the command and creates a pipe to the process running the command. This is done by tools like the man page browser and search. Output from the man or grep commands that they issue is parsed by the DEC FUSE tool, often using a simple mechanism such as the standard C library function fscanf, which applies a format string to a line to parse it. Some tools also make use of lex with or without yacc to aid in parsing the output of the commands.[12,13]

Other tools use PMAT (pattern matching) routines for examining command output for desired patterns. The PMAT functions were developed by Steven Reiss as part of the FIELD environment. They are used in FIELD both for managing messaging as well as for interpreting the output of UNIX commands. For DEC FUSE interfaces to UNIX commands, the patterns used by the PMAT routines are organized in tables. Portions of two of these tables are shown in Figure 7. These examples are for the output of gnumake and a make program supplied with Digital UNIX.[14] For this analysis, there are two significant parts of each pattern table entry: a text pattern that may be found in the command output, and the name of a routine to be called if the associated pattern is found. For example, when the error message "Failed to remake target file '%1s'" is recognized, the function named make_giving_up is called with arguments that match specifications in the pattern string.

Additional values from the table (omitted in the figure) are also passed as arguments to the routine. The string '%1s' in the pattern is similar to the conversion specifications used by scanf. It represents a field in the output that will be passed to the recognition routine when a pattern is recognized. Some of the field specification characters used are given in Table 2. The number preceding most field specification characters tells the pattern match what position this field should hold in the argument list passed to the recognition routine. When there is no number with a field specification character, that field is not passed to the recognition routine.

Choosing the Appropriate Command Interface Method

The DEC FUSE product was designed to be portable across several hardware platforms and many operating system versions. DEC FUSE was developed on the ULTRIX system and has been ported to SunOS, AIX, HP-UX, and Digital UNIX operating systems. It was released to customers on all these platforms, except AIX. Since portability across platforms and versions is a goal, interfaces for different command implementations and versions need to be considered. The choice of interface method is made based on the complexity of the interface (the number of commands and expected responses), the number of different interfaces needed because of system differences, and the rate at which the interfaces are evolving.

Most common UNIX commands, such as grep, man, and diff, have regular output that seldom changes. The versions of these commands on the desired platforms and operating systems have few differences, so it is not difficult to write portable code that can issue these commands and interpret the output using the lex, yacc, or the scanf functions.

In cases in which the output is less regular and varies across commands and platforms, the PMAT facilities are more appropriate. This includes the DEC FUSE builder, which must support several different make programs on the supported platforms. The PMAT facilities allow for interpreting a large number of different format lines and for selecting tables of patterns appropriate to the underlying command. This makes it easier for the builder to accommodate a variety of make programs and interpret both output from make and output from compilers.

```
/****** Pattern table for gnu make ******/
static MAKE_PAT gnu_pattern_table[] = {
   {"Reading makefiles...",          gnuscan_makefile, ...},
   {"Considering target file'%1s'",  gnuscan_consider, ...},
   {"Found an implicit rule for'%1s'", gnuscan_flags,    ...},
   {"Updating goal targets....",  gnuscan_makefile, ...},
   {"File'%1s' was considered already",  gnuscan_done,     ...},
   {"Must remake target '%1s'",          gnuscan_flags,    ...},
   {"Failed to remake target file'%1s'", make_giving_up,   ...},
   {"No need to remake target '%1s'",     gnuscan_flags,    ...},
   {"# Files",                     gnuscan_files,    ...},
   {"# Not a target:",               gnuscan_notarget, ...},
   {"#  commands to execute",           gnuscan_setrules, ...},
   {"#  Phony target",                gnuscan_defflags, ...},
   {"#  Precious file",               gnuscan_defflags, ...},
   {"# VPATH Search Paths",              gnuscan_files,    ...},
   {"# gnumake: Entering directory'%1s'", gnuscan_proj,     ...},
   {"# gnumake: Leaving directory'%1s'",  gnuscan_proj,     ...},
   {"%1s: %2r",                    gnuscan_def,      ...},
   {"%1s:",                      gnuscan_def,      ...},
   ....
};


/****** Pattern table for dec make ******/
static MAKE_PAT dec_pattern_table[] = {
   {"doname(%1s,%2d)",                decscan_consider, ...},
   {"setvar: @ = %1s noreset",        decscan_flags,    ...},
   {"setvar: ? = %1r",              decscan_flags,    ...},
   {"! = %1r",                     decscan_adjust,   ...},
   {"look for explicit deps. %1d",   decscan_flags,    ...},
   {"look for implicit rules. %1d",  decscan_flags,    ...},
   {"Current working directory for make is %1s",
                                   decscan_proj,     ...},
   {"%1s: %2r",                      makescan_def,     ...},
   {"%1s:",                        makescan_def,     ...},
   {"Reading %1s",                   decscan_makefile, ...},
   .....
};
```

Figure 7
make PMAT Patterns

Table 2
Some PMAT Field Specification Characters

Field Character	Data Type
d	Decimal number
x	Hexadecimal number
c	A single character
s	A string, delimited by white space
q	A string, delimited by quotation marks
r	A string, from the current location to the end of the line
e,f,g	Floating-point numbers

The tool with the most complex command interface is the debugger. The debugger shares the following issues with other tools, but demonstrates them most forcefully:

1. Debuggers are big and complex. Debuggers are more complex than the commands used in other DEC FUSE tools. Each debugger engine accepts many commands, all of which have their own output that must be parsed. The debugger engine also continues to run while the user works. Unlike most other tools, the debugger engine is not restarted every time the user wants more information, so the debugger process must be managed over a long period of time.
2. Debuggers are evolving more quickly. Debuggers frequently change to support new needs (for example, new languages like C++, threads, or hardware architectures), so new debugger commands or new output from old commands can be expected often.
3. Synchronizing the front end and the debugger engine is a complex task. The graphical front end

must remain synchronized with the debugger engine it is running. Preserving this synchronization is made more difficult for three reasons. First, users can enter debugger commands directly as text, making it difficult for the front end to determine their effect. These commands may require updates to the graphical displays or the internal state information used by the front end. Second, the debugger may not be in a state where it can accept commands (when the user program is running for example), so the front end cannot update displays. Third, spontaneous and unexpected debugger engine output may occur as the result of traces or certain breakpoints.

4. Different debuggers use different commands. Commands on different debuggers can be different in both name and design. For example, with the dbx debugger available on SunOS, AIX, and Digital UNIX, the commands func and file can be used to find the currently active function and the name of the source file where that function is defined. The xdb debugger used on HP-UX, however, uses the L command to present both the current function and the name of the file where it is defined, as well as to display the current source code line.
5. The same debugger commands have different output. Other commands, although similar in name and design, can produce output that is different enough to cause problems. One example is the where command used in dbx on both Digital UNIX and SunOS platforms. This command returns the current stack information. The Digital version includes a pointer character (>) to show which stack entry is the current scope; however, the SunOS version does not supply this scope information. Therefore, a debugger GUI program must be carefully designed to get needed scope information if it must support both debugger engines.
6. The output of some debugger commands is complex, and the results of some debugger commands are difficult to parse. For example, in the display of the content of a data structure, the format of the output will vary depending on the source language used in the application.

Experiences with DEC FUSE suggest that there is no easy solution. Addressing these issues results in many specialized routines in the DEC FUSE debugger tool to both construct debugger commands and interpret the results. Techniques that help to make the problems more manageable include the following:

- Cleanly separate generic-GUI and command-specific code. The design of the debugger GUI identifies the operations that it requires of the debugger engine and the data that it must get from the engine. These are provided by a set of functions whose implementation will vary from one engine to another. These functions will be modified over time to accommodate the evolution of the engines. Another method being designed now is to use C++ classes to encapsulate code for each supported debugger engine.
- Limit the details that the GUI depends on. One way to limit the dependency of the GUI on the details of the engine is to provide GUI support for only the most frequently used debugger operations, while providing a command interface for the remaining operations. Another technique is to avoid interpreting the output of the engine when possible and simply display the output of the command in a text window.
- Implement special interface commands in the engine. When it is possible to change the underlying debugger, special commands and output can be implemented by the debugger designed exclusively for use by the GUI front end. For example, the DECladebug debugger engine has been modified with the introduction of two new commands for use by the graphical interface that simplify the task of displaying data structures in the GUI. Although other commands display data structures for the user, the format of the output of these commands is designed to be easily interpreted by the GUI. These commands are designed for the exclusive use of the GUI. They need not be changed for the user, for example, to improve readability; thus the evolution is controlled.

Fortunately, most UNIX tools are not as complex as the debugger. In fact, building a GUI for commands with output that seldom changes and is consistent across implementations is a straightforward task.

Using Messaging to Make Independent Tools Work Together

As described earlier, each DEC FUSE tool focuses on a single, separate software development task. This design philosophy, sometimes called "divide and conquer," combined with the DEC FUSE multicast messaging system (MCMS) makes it easier to maintain or replace tools. DEC FUSE tools can therefore be easily replaced with alternative tools that provide the same function.

MCMS is the key to making independent tools work together. Any message sent by a tool is delivered to all tools that express an interest in receiving the message. Some messages, called notifications, are defined to have no response. Other messages, called requests,

have responses for which the sending tool usually waits. A tool can also eavesdrop on requests that will be handled by other tools. A DEC FUSE component called the DEC FUSE message server keeps track of the active tools and which messages each can send and receive.

Messaging with MCMS

Messages used by tools are easily defined in a TIL file, written in the DEC FUSE tool integration language. An example is the manager.til file used by the DEC FUSE code manager. Part of manager.til is shown in Figure 8. Each TIL file can define one or more tool classes. Each class definition describes how a single DEC FUSE tool will be integrated with the rest of DEC FUSE. A class definition contains three parts:

1. Attributes: This is a collection of tool attributes such as the string to be used in the DEC FUSE tools menu and the command to invoke the tool.
2. Messages: This section lists definitions for all messages sent and received by the tool, including their arguments and return values. Messages that have return values defined are called requests, and the returned value is expected by both the message switch and the tool that sent the request. Messages with no return value (the type is void) are called notifications. The keyword trigger is used if the message should automatically start the tool.
3. States: This section describes when each message may be used during the execution of the tool. This section defines one or more states in which the tool

```
class MANAGER = {
      Attributes {
            label    = "Code Manager";
            accel    = "Meta+M";
            path     = "$(FUSE_SH_BIN)/manager";
            .... };

      Messages {
      /* messages accepted by the FUSE code manager */
      char *ToolReconfigure(char *working_directory,
            char *target_directory, char *target, char *other);

      trigger char *CheckIn (char *libraryname, char *filename,
            char *revision, char *comment, int  keepfile,
            int  filemode);
      ....

      /* messages sent by the FUSE code manager */
      void   CheckInNotification ( int   instance_id,
            char *libraryname, char *workdir, char *filename,
            char *revision, int status);
      ....  };


      States {
            start {
                  receives {
                  ToolReconfigure,
                  ....  };
                  sends {
                  ....  };
            };
            running {
                  receives {
                  ToolReconfigure,
                  CheckIn,
                  CheckOut,
                  .... };
                  sends {
                  ToolReconfigure,
                  CheckInNotification,
                  .... };
};    };    };
```

Figure 8
DEC FUSE Tool Integration Language File

may exist. Tools can change their state, and within each state only the listed messages may be used. Most DEC FUSE tools need only two states: an initialization or start state used during tool start-up and a running state. Other states may be needed by some tools. For example, the builder uses a building state to advise the message server that a build is in progress and that some requests (like another build request) are not allowed.

A TIL compiler translates the TIL files of DEC FUSE tools into the data files needed to run DEC FUSE. Figure 9 summarizes how the files generated by the TIL compiler for a DEC FUSE tool (named fuse_tool) fit into the architecture of DEC FUSE.

The TIL compiler combines information from the fuse_tool TIL file with TIL files for tools already installed on a system. The TIL compiler generates three files:

1. fuseschema.msl – This file tells the message server which tools wish to receive which messages.
2. tools.rc – This file tells the control panel how to start each tool. Tools may be started in response to a trigger message or manually from the Tools menu found in each DEC FUSE tool.
3. FUSE_fuse_tool.c – This file contains functions for each of the messages that the tool wishes to send. This file is compiled and linked with fuse_tool along with libfuse.a. Messages are sent by simply calling these functions. This file also contains an initialization function in which callback functions for messages that the tool receives are registered.

The use of the TIL compiler in DEC FUSE provides a mechanism similar to a remote procedure call facility. This allows tools to send a message using a single function call. This contrasts with the messaging mechanisms used in the HP SoftBench and Sun SPARCworks products, which require a number of calls to the messaging application programming interface (API) to allocate, assemble, send, and free a message. These mechanisms also require tools to assemble and register patterns corresponding to the messages that they want to receive, a function handled by the initialization function in the C source file generated by the TIL compiler.

To simplify the task of integrating tools, DEC FUSE also supplies a DEC FUSE message monitor. This tool monitors and debugs messages sent by tools and provides a mechanism for integrating shell scripts as tools that can send and receive messages.

Simplified Tool Replacement

MCMS does not require the user to specify the tool that does the work. When a tool sends a message using MCMS, it does not specify what tool should service the message. This allows for replacement of the tool that services the messages with an equivalent tool, without making any change to the sender. This mechanism is used in DEC FUSE to allow users to select which of three editors they want to use and whether they want to use a GUI debugger based on dbx or DECladebug.

This mechanism also facilitates upgrading the DEC FUSE environment. Recently, the Motif help widget in DEC FUSE was replaced with the HyperHelp tool. The replacement was facilitated by continuing to use the existing messages. This isolated all changes to the DEC FUSE help tool. The help tool continues to receive messages of the form

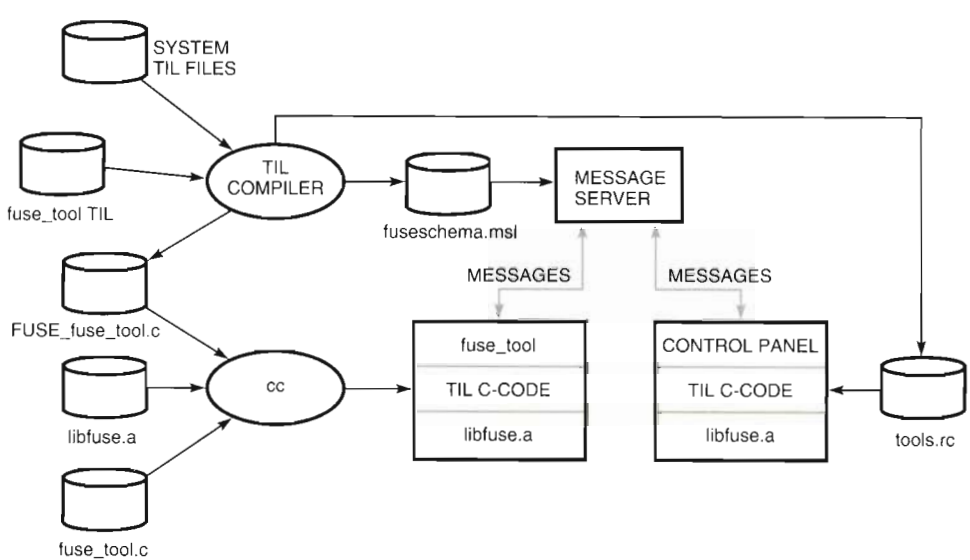


Figure 9
Use of TIL-generated Files in the DEC FUSE Architecture

```
trigger void HelpShowTopic(char *product,
                           char *mode,
                           char *topic);
```

In the previous version, the message argument, topic, was a string that identified what kind of help was desired. The new help tool uses numbers instead of names to identify help topics. Consequently, a simple mechanism was designed to translate the strings received in the HelpShowTopic messages to the desired HyperHelp topic number.

Conclusion

DEC FUSE provides an integrated programming environment for UNIX software development that takes advantage of the graphical capabilities of workstations. Two key techniques are used to implement DEC FUSE:

- The layering of GUIs on existing UNIX command line tools
- A multicast messaging mechanism that permits tools to interoperate without limiting the environment to specific tools

The GUIs provide point-and-click interfaces for invoking operations and specifying options and use pictures and diagrams in addition to text to display information. At the same time, the use of traditional UNIX commands to perform programming tasks preserves the user's investments in those underlying tools.

The GUIs interpret the output of UNIX commands and present the information in pictorial and interactive displays. A variety of techniques can be used to process the output of a command line tool, depending on the complexity of the tool output. Simple text-processing techniques are usually adequate for interpreting the output of command line tools. When the underlying tool output is syntactically complex or evolving, or when considerable state information is frequently needed from the underlying tool, it becomes difficult to apply these techniques. Under these conditions, designs that avoid the processing of human readable output are preferred.

The use of messaging is consistent with the UNIX philosophy of creating simple tools and letting the user combine them in any way that might be useful. The messaging mechanism ties the individual tools together into an integrated environment by allowing tools to invoke operations in other tools on the user's behalf. This eliminates steps for the user, and it also eliminates the potential for errors. Because the tools are still autonomous and interface solely by means of the messaging, equivalent tools that accept the same messages can be substituted, allowing for user and project preferences.

Acknowledgments

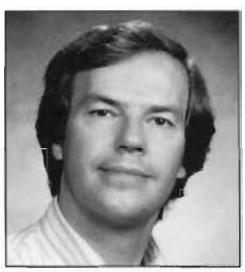
We would like to thank the many past and present members of the DEC FUSE team who contributed to the design and implementation of the DEC FUSE product. We also want to acknowledge the work on FIELD done at Brown University by Steven P. Reiss and his students that laid the groundwork for DEC FUSE and other software development environments.

References

1. B. Kernighan and R. Pike, *The UNIX Programming Environment* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984).
2. R. Stallman, *GNU Emacs Manual* (Cambridge, Mass.: Free Software Foundation, 1988).
3. R. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics,* vol. 5, no. 2 (April 1986).
4. *DEC FUSE Handbook* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q8ZMA-TE, 1994).
5. *DEC FUSE EnCASE Manual* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q8ZPA-TE, 1994).
6. M. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal* (June 1990): 36–47.
7. *Common Desktop Environment: Getting Started Using ToolTalk Messaging* (Mountain View, Calif.: Sun Microsystems, Inc., 1994).
8. S. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development* (Boston: Kluwer Academic Publishers, 1995).
9. S. Reiss, "Interacting with the FIELD Environment," *Software—Practise and Experience,* vol. 20 (June 1990): 89–115.
10. S. Reiss, "Connecting Tools Using Message Passing in the Field Environment," *IEEE Software,* July 1990: 57–66.
11. D. Zaremba, "Adding a Data Visualization Tool to DEC FUSE," *Digital Technical Journal,* vol. 7, no. 2 (1995, this issue): 20–33.
12. M. Lesk and E. Schmidt, "Lex—A Lexical Analyzer Generator," *Computer Science Technical Report No. 39* (Murray Hill, N.J.: Bell Laboratories, 1975).
13. S. C. Johnson, "Yacc: Yet Another Compiler-Compiler" (Murray Hill, N.J.: Bell Laboratories).
14. R. Stallman and R. McGrath, *GNU Make—A Program for Directing Recompilation* (Cambridge, Mass.: Free Software Foundation, 1993).

Biographies

Richard O. Hart
Rich Hart joined Digital in 1980 and is currently a member of the FUSE Group. Prior to his work on the DEC FUSE programming environment, Rich was a member of UEG (ULTRIX Engineering Group) and led the first version of the Palladium distributed printing project at MIT's Project Athena. As one of Digital's representatives to the X/Open, POSIX, and ANSI standards groups, Rich has contributed to the development of software standards for transaction processing, printing, and CASE environments. He earned a Ph.D. from the University of Connecticut and is a member of ACM and IEEE.



Glenn Lupton
Glenn Lupton is a consulting software engineer and has been with Digital for 20 years. During this time, he has worked primarily on programming environments and tools, including Bliss compilers and DECset. For the last two years, he has been the technical director of the DEC FUSE project with responsibility for the overall technical content of DEC FUSE. Glenn received B.S.E.E. and M.E.E.E. degrees from Rensselaer Polytechnic Institute.

Donald A. Zaremba

Adding a Data Visualization Tool to DEC FUSE

Digital's Data Visualizer tool uses condensed file views to display thousands of lines of source code. These displays can include the output of many other tools. As part of the DEC FUSE programming environment, the tool helps software developers by providing capabilities for displaying large bodies of text with associated events or statistics. The Data Visualizer tool combines the results of other tools into a single display, keeps track of work items, and scales up to support large software projects.

In January 1993, Digital began research on a tool for visualizing large sets of data. The design of the Data Visualizer tool was complete in March 1995, and the tool is scheduled for inclusion with the next major version of the DEC FUSE software. DEC FUSE is a programming environment for UNIX that provides an integrated suite of graphically oriented tools built on the commonly used UNIX programming tools. For more information on the DEC FUSE environment, see the paper "DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools" in this issue.[1]

In this paper, we focus on the technology that was used in the data visualization tool and the process by which this tool was taken from an advanced development project to become a part of an existing product. We start with a discussion of the problems encountered when visualizing large sets of data, the various graphical techniques that are used to solve these problems, and the implementation of these techniques in a demonstration tool. We then describe the design of the final tool, its evolution from the prototype into a product, and its integration with the other DEC FUSE tools. We then give a functional overview of the tool and scenarios of how it can be used. We conclude with comments on the process from advanced development work into final product.

Development of a Data Visualization Tool

Software development of even a moderately sized project typically involves working with many files and hundreds of thousands of lines of source code. Working with so much data in so many files is difficult because most software tools are written to work on a single file at a time (like a compiler or an editor). Those tools that do operate on multiple files (like a grep tool used with wildcards) produce a stream of output that can be large and that can only be associated with the source code by identifying a line number or by displaying a single line of source in context. Although these tools do provide the requested answer, they provide little of the context that would help the user see how this answer relates to the source code or how it would relate

to other answers. It is often hard to see how these detailed answers fit into the large picture.

One technique for solving this problem is to use computer graphics in the display portion of software development tools. Graphics are used to display information such as build dependencies, cross-reference data, call tree data, and class hierarchies.

Unfortunately, when the application becomes large, the graphic displays become too dense to provide any real insight into the relationships between the components in the application. The screen is simply not large enough to display all the information. The layout of nodes on a two-dimensional display is often inadequate to effectively represent the complexity of the underlying structure and relationships in the code. The common use of overlapping windows of data actually hides data, preventing users from seeing important relationships among the windows or even knowing which windows contain relevant data. In effect, programmers who must work on today's complex software applications are confronted with a situation similar to entering a large dark room with a complicated piece of machinery in it. Current technology hands the engineers a penlight and says figure out what the machine is, how its parts work, and then make enhancements to it.

The Data Visualizer tool addresses some of these problems by providing a condensed view of source code; the tool is capable of displaying thousands of lines of code in a single view. This condensed display is used as a backdrop for showing the output from tools and how it relates to the source code. Figure 1 is a sample screen output from the Data Visualizer tool being used in conjunction with a search tool to find occurrences of a particular string. This simple example shows many of the features of the Data Visualizer. The rendering of each file in the view shows the indentation of the source code. Source code is colored to show comments in green, the beginning of functions or procedures in red, and the actual code in gray. The sizes of files and functions are readily apparent. The results of the search inquiry are highlighted.

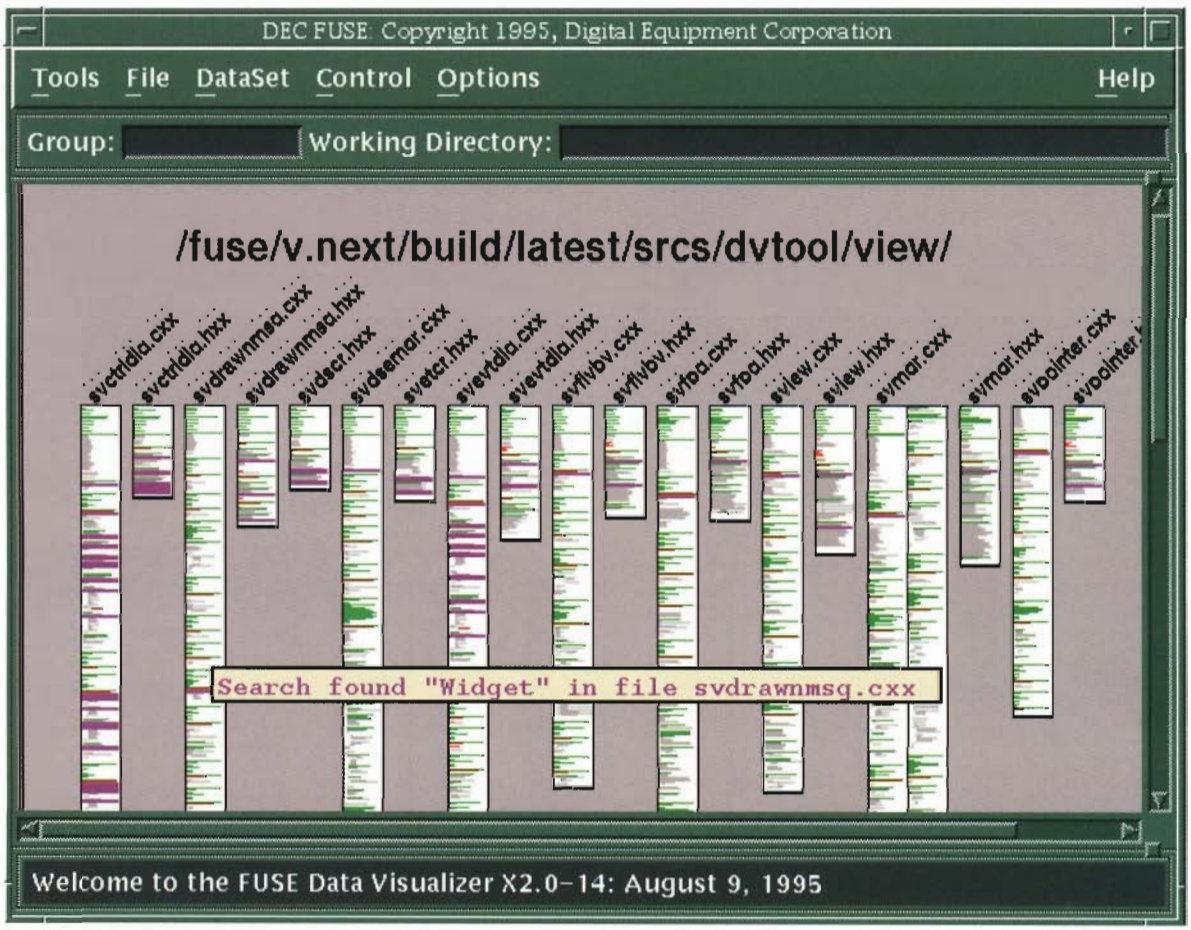


Figure 1
Main Window of the Data Visualizer

Graphical Techniques

During the early phases of this work, research was done to find appropriate graphical techniques. This section describes in detail three techniques that influenced our design and appear in some form in the Data Visualizer tool. It also gives references to related work.

Condensed File View

One technique that looked promising from the very beginning was the condensed file representation done by Stephen Eick in 1993. In his paper "Graphically Displaying Text," he describes a program called SeeSoft that is used to display statistics associated with lines of text.[2,3] He has used this technique to show statistics about lines of program source code and other text files, such as text from the Bible or revision history of text paper. He also uses the technique to analyze computer log files and describes that work in a separate paper.[4]

The idea behind the SeeSoft program is to create small pictures of files that reveal information about a file in a nontextual manner. The size of the rectangle is scaled to the number of lines in the file. Each line of text is shown with the correct indentation and length. In addition, lines can be color-coded either to reveal program structure or to highlight some point of interest. As an example, green lines could be used for comments, red lines to indicate the start of each function, and gray lines for executable code. As can be seen in Figure 2, the information reveals the size of each file and some information about the file contents. It is easy to see where function definitions begin, because the red lines stand out. Also, the indentation of the code helps the viewer recognize programming structures like if then else statements or case statements.

One of the appeals of this method was the ability to display many lines of source code. (Eick's SeeSoft tool claims to display as many as 50,000 lines of code.) Programmers can get a clear and complete overview of their code. From the simple view shown in Figure 2, with no additional data, we can see the size of each file, the relative size of individual functions in a file, and the frequency and distribution of comments.

Multiple Levels of Details

We investigated a second technique that seemed appropriate: the drawing of objects in multiple sizes and in multiple levels of details. The concept of adjusting the amount of detail presented to the user as a function of the apparent size of an object is a technique developed in a unique computer interface model called Pad.[5] Pad provides an infinite two-dimensional information plane that the user can browse using portals (analogous to magnifying glasses) to zoom into the data.

The larger the object, the more details are revealed. This corresponds to the notion that things that interest us are the ones we bring closest to us; they require the greatest amount of detail. Those items of lesser interest are placed in the background and drawn smaller. As can be seen from the pictures in Figure 3, as the size of the file increases, more details are shown about the file. The smallest picture reveals only the major structural parts of the file; we call this *chunk* level. Each chunk is drawn as a colored rectangle and represents either a group of comments (green), the start of a function (red), or lines of executable source code (gray). The next picture shows line-level detail like that shown in Figure 2, and the last picture shows each line large enough to be drawn as readable text. Note also that the largest picture begins to look like a text editor and that the scroll bar on the right is a chunk-level rendering of the file.

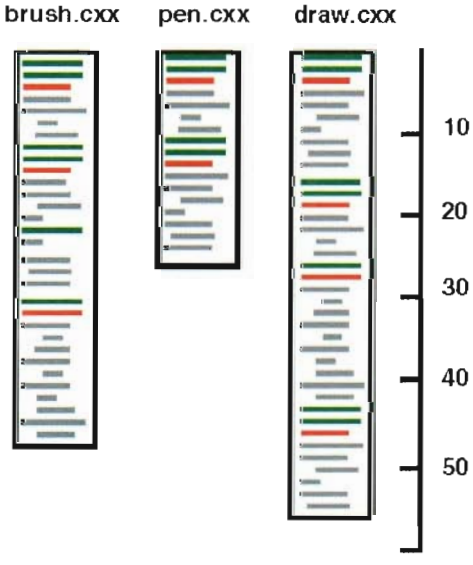


Figure 2
Condensed File View

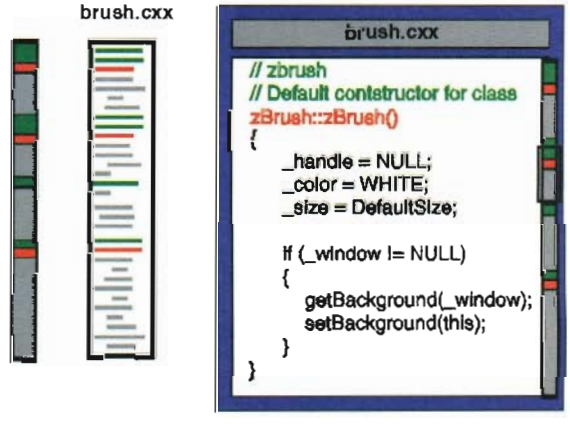


Figure 3
Multiple Sizes of Files

The Use of the Third Dimension

We also chose to investigate the use of the third dimension for ways to better visualize large, dense graphs. We did not pursue this work for several reasons, which we describe later in this paper.

We did find a simple use of three-dimensional (3-D) viewing that was beneficial when trying to visualize certain types of data. We converted the condensed file pictures into 3-D views by adding a small side to each picture. We could use that area to show line-related data as in Figure 4. This example shows a numeric value (the blue lines) associated with a line of source code. The horizontal dotted line is a threshold, and values that exceed the threshold are drawn in red. We use this type of graphic to show source code profiling data, like execution counts and CPU time. Even though it is a simple drawing, it uses a 3-D effect that helps the user visually organize a great deal of information. It is relatively easy for a user to look at the front data at one moment and put the side data off into the background, and then change focus and examine the side data. The effect is even more noticeable and useful when many of these 3-D file pictures appear in the same display. An example of this is given later in the section on the SoftVis Program.

The Advanced Development Project

This section describes the advanced development phase of the project. It discusses the process used, the software prototypes produced, and the major design decisions made during this phase.

The Advanced Development Process at Digital

The type of work done in Digital's Advanced Development Group, working with new technologies and implementing new ideas, is difficult to do within a schedule-constrained product development organization. Although the goals of advanced development work may be well specified, only a vague idea of a possible solution and of the time needed to find the solution is known. These two facts make it impossible to schedule advanced development work in a product's project plan. At Digital, the Advanced Development Group is a separate organization that operates outside the product schedule constraints of other groups. It is staffed by engineers from the development groups, who rotate into the Advanced Development Group, perform their work, and then return to their sponsoring group to transfer the technology into a product.

The stated goal at the beginning of our project was to enhance the software browsers available in the DEC FUSE product by adapting the results of current research in visualization techniques. Of particular interest was the ability to browse large software systems containing large amounts of source code. We were also looking for techniques that would provide new information about source code and new ways of looking at source code. Our objective was to add features to DEC FUSE that were not currently available in other products.

The process we used was to research as many different techniques as possible and select those that appeared most promising for prototyping. The prototypes gave us experience in the technology and helped us in our evaluation. We then sought input from our sponsoring group to determine which prototypes were feasible to add to the product, and we continued to develop and refine these.

Using 3-D Computer Graphics

At the beginning of the project, we wanted to explore the 3-D graphics technique. For this research, we used a DECstation 5000/20 workstation with a 3-D graphics accelerator option installed. The code was written in C++. We used the Motif standard to build the windows and menu part of the user interface and the programmers hierarchical interactive graphical standard (PHIGS) to write the 3-D graphics code.

We quickly built three demonstration programs to gain experience in 3-D graphics programming. The first program was an instrumented C++ class library that created and destroyed color-coded cubes in 3-D space as constructors and destructors were called. Message passing was shown by connections between the cubes. The z-axis was used for time: the older an object became, the farther back it would appear on the z-axis. The second demonstration drew hierarchies in 3-D space and gave the user limited capabilities for manipulation in 3-D. The third demonstration visualized a C++ class as a cube in 3-D space, with different sides being assigned different types of data. One side

Figure 4
3-D File Picture

contained a class inheritance graph, another contained a condensed view of the interface to the class, and the third side contained a window into the source code of the class.

After a short period, for several reasons, we stopped working with 3-D graphics. We realized that the types of visualizations we were doing would require 3-D accelerators on users' workstations, and we knew that would not be acceptable. In addition, development of this technology would take a great deal of time, and we felt we could make better progress working on other graphics techniques.

Early Prototypes

Having seen the work done by Stephen Eick, we decided to experiment using his technique. We also started to think about the concept of building a framework that we could use to build prototypes of different techniques. Eventually, this evolved into the design we describe later in this paper. At this time, we also considered what platform to use. Our sponsoring group had developed the DEC FUSE product for the UNIX environment, but other groups were starting to work on the Windows NT operating system for personal computers. Since we were interested in learning more about the Windows programming environment, we decided to produce code that would work on either platform and to build prototypes on both platforms. In hindsight, our decision to support multiple windowing systems was a diversion that did not directly contribute to the project goals, but it was a valuable learning experience.

To achieve cross-window system portability, we developed a class library that encapsulated parts of the programming interfaces on the MS Windows system and the X Window System. We decided to restrict our class library, collectively referred to as the "ZWindow" or "ZWIN component," to encapsulate only the low-level graphics drawing routines (e.g., line and rectangle) and avoid trying to encapsulate all the graphical interface components like windows, icons, and menus. We encapsulated at the level of the graphics device interface (GDI) on MS Windows and the X library interface (Xlib) on the X Window System. This worked well; we achieved portability of our graphics drawing code, which was our area of concentration. The fact that we had to do separate implementations for the remainder of our user interface (that is, the menus, toolbars, and dialog boxes) was not a hindrance since the bulk of our code was still portable.

Designing the ZWIN interface was fairly straightforward. The line and shape drawing routines were easy to encapsulate because they existed on both platforms. The drawing contexts were different. The MS Windows system has color pens and brushes to control drawing attributes; but on the X Window System, all drawing attributes are defined in a single data structure, the graphics context (GC). We decided to create classes for pens and brushes and to handle the X Window System implementation by encapsulating an appropriate GC in the pen and brush classes. The largest class in the ZWIN component was the canvas class. It encompassed a DrawingArea Widget on the X Window System and a window on MS Windows. It had member functions that provided all the drawing functions available (e.g., line or rectangle), as well as functions to select the appropriate drawing object (pen or brush).

The condensed file view was implemented in two sets of classes. A set of file-type–dependent scanner classes was developed to handle the parsing of C, C++, Ada, makefiles, etc. Once scanned, a single file visualization class could perform the rendering of the object on the display. Speed was a concern since we wanted to be able to visualize an entire directory of files very quickly. To do this, we wrote a small, efficient scanner for each type of file that could pick out only the relevant information as quickly as possible. Throughout our work on all the prototypes and into the final product, we found that we could always fill a complete display without any noticeable delay to the user.

Figure 5 shows part of the first prototype. It displays a condensed file view of all the text files in the default directory. Files were sized to fit within the size of the window, with an appropriate level of detail shown. Files could also be individually selected and resized. Files are shown in the three different levels of detail described in Figure 3. Most of the files are drawn at the chunk level and reveal only the relative size and location of each function in the file. Two of the files have been enlarged to show line-level details, and one file has been fully enlarged to be a readable size.

Later prototypes improved upon the design of this condensed file view. We also implemented other views that we thought would be useful. The C++ class view rendered a condensed picture of a C++ class with its member functions and data members. It is described later in this section.

SoftVis Program

Throughout the process of creating the first few prototypes, we kept in mind the concept of building a framework that we could use to speed up the delivery of new graphical techniques. The SoftVis demonstration program used that design. Based on a View-Object-Tool architecture, its concept was that a view would set the backdrop and style for the display, such as the condensed file view. We would render objects into that view style and support many different types of objects per view. Tools would then be written to

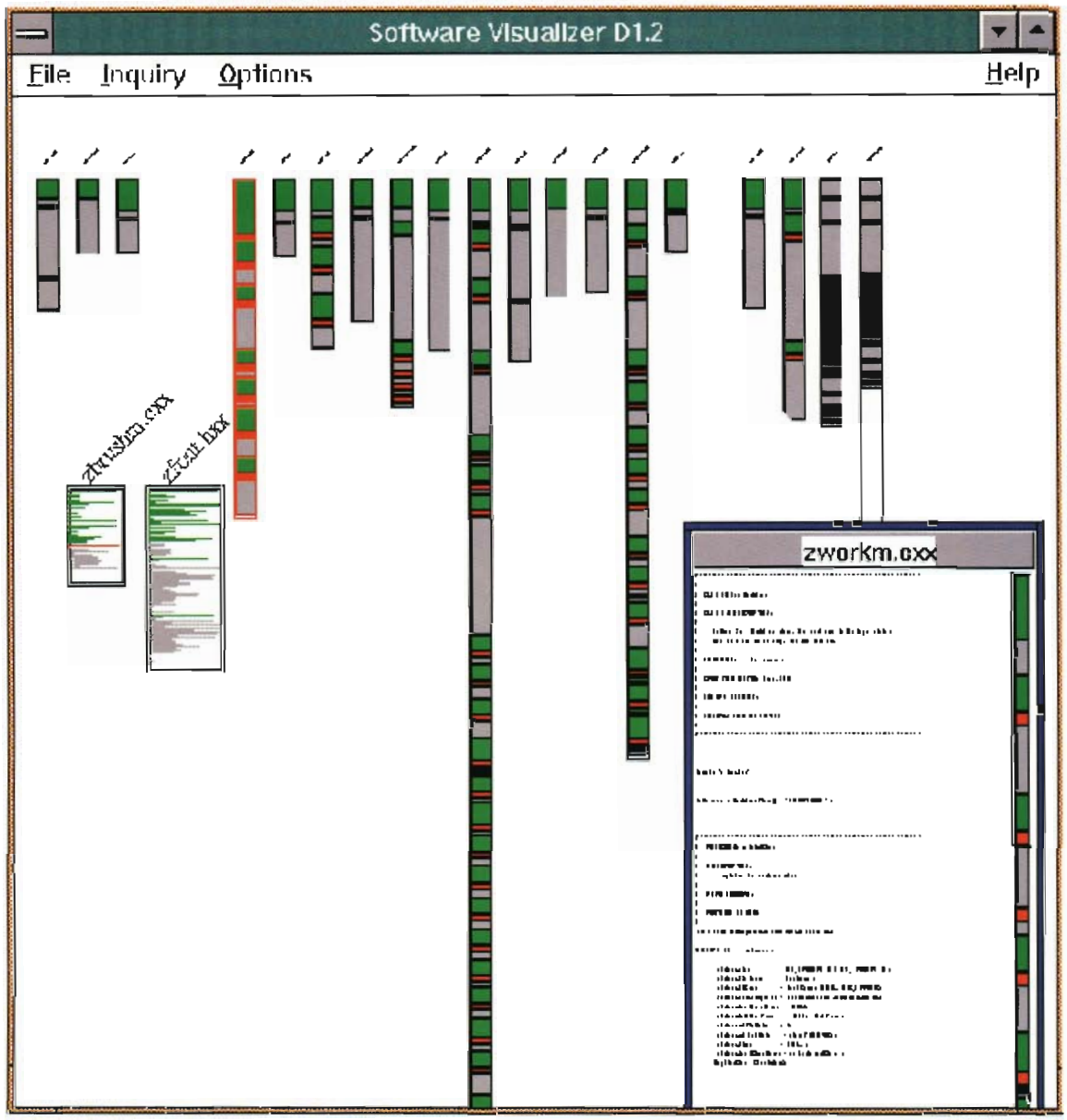


Figure 5
First Demonstration Program

interact with the objects in the view. Our objective was to develop a "plug-and-play" architecture that supported the following:

- View
  - Condensed file view
  - Condensed file 3-D view
  - C++ class view
- Object
  - C++ source code
  - C source
  - Ada source
  - .o (object files)
  - .a (library files)
  - executable files
- Tool
  - Magnify tool
  - Probe tool
  - Cross-reference tool
  - Search tool
  - IF-DEF lens tool

The goal was to be able to create a view containing all the files in a directory and displaying an appropriate visualization for each of the file types (either a text file or a binary file), and to enable the tools to operate on all the objects in the view. For example, the magnify tool would show a readable view of the text in a source file; however, when used on a binary object file, it would show information about the size, address, and type of segments in the file.

Figures 6 and 7 are screen captures from the prototype. Figure 6 shows a cross-reference tool being used on C++ source files. The list box shows functions from all the source programs, and the highlighted function color-coded lines point to where that function is first declared, implemented, and called. Figure 7 shows the magnify tool used in the 3-D file view to show source code details and profiling data. In this case, the profiling data is a mock-up of line execution counts; the real tool will use this space to report actual data.

Figure 8, also a screen shot from the prototype, shows the C++ class view. This view uses a condensed representation of a C++ class. Each line in the class corresponds to either a member function or a data attribute of the class. These are grouped together as public, protected, and private members. Member functions are shown in red; data elements are shown in blue. Inheritance is shown by connected arcs.

SoftVis Design

The system is divided into several components. Each component can be built separately; has its own makefile; and, in most cases, its own test programs. Table 1 gives an overview of these components and their relative sizes as of the latest base level.

The SoftVis design begins by supporting the desired prototype architecture of View-Object-Tool. A component was developed for each of these; it contained a base class, derived classes, and supporting classes.

From Advanced Development to End Product

This section describes the effort required to turn parts of the final advanced development prototype into a product-quality tool for release with DEC FUSE.

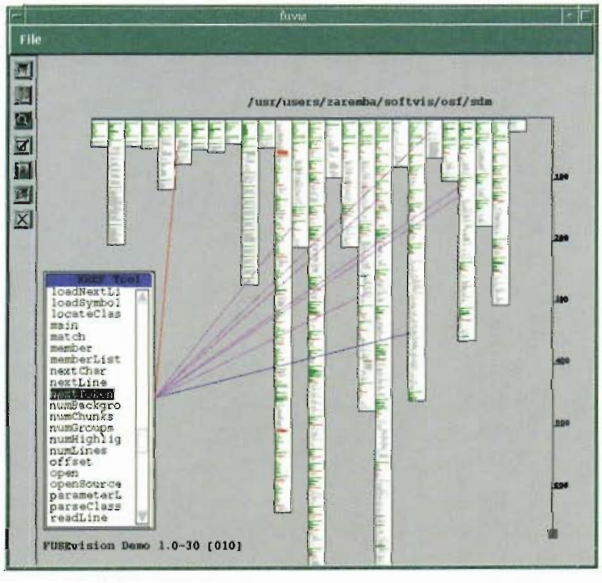


Figure 6
Demonstration of the Cross-reference Tool

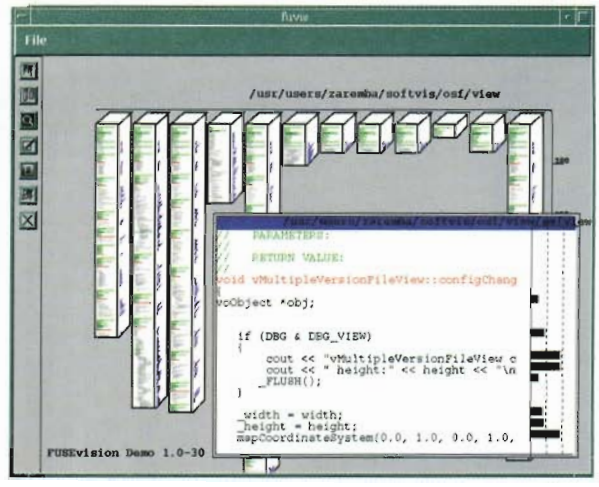


Figure 7
Demonstration of the 3-D View with Profiling Data

Finding a Place for the Work

At the conclusion of the advanced development project, we returned to our sponsoring group and attempted to introduce the data visualization technology into the product. A number of obstacles had to be overcome: The SoftVis program was written in C++, and DEC FUSE had been written almost entirely in C. The requirements for the next release of DEC FUSE had been gathered, and the goals were set. Where exactly would the new data visualization technology fit into the DEC FUSE product set?

At first we tried to build a class of reusable software components that DEC FUSE tools could use to incorporate the new technology. This would be a set of Motif widgets that encompassed the techniques prototyped in the SoftVis program. Although progress was made on building the widgets, no progress was made incorporating these into any of the DEC FUSE tools. Their incorporation would have required major changes to the user interfaces of these tools, and it was not clear that the benefits would justify these changes.

In hindsight, we realize that the plug-and-play design we used for the prototype did not match the DEC FUSE design of loosely coupled separate tools that passed data by means of simple messages. Although the plug-and-play approach made it easy to add new components into the model, its tightly coupled design made it difficult for us to take parts of that design and use them in the DEC FUSE product.

The proposal that was finally accepted was to develop a new, separate tool, called the Data Visualizer, that

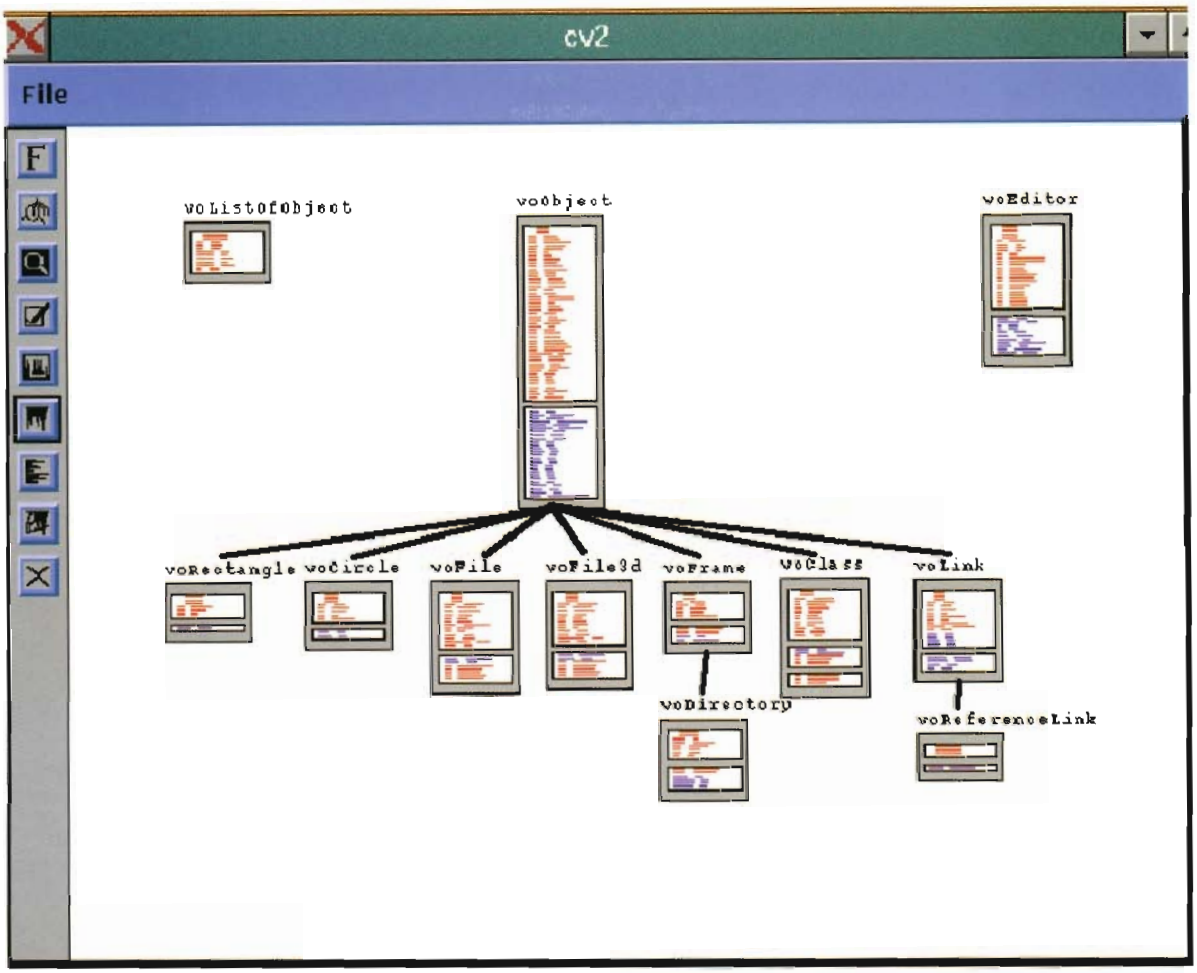


Figure 8
Demonstration of the C++ Class View

Table 1
Components in the Prototype Design

Component	Description	Lines of Code	Classes
VO	Base classes, voObject, and voEditor. Also, voFile class and other classes derived from voObject. Implements features for selecting, moving, resizing, and drawing objects.	5,000	10
TOOL	Base tool class, voTool, and classes derived from it. Includes voLens, voProbe, voMagTool, and voXRefTool.	2,500	10
VIEW	The vBaseView class is derived from voEditor. The three main views of the tool are then derived from vBaseView. The main views are vFileView, vFile3dView, and vClassView. This component also contains executable test programs for each view.	2,400	4
SDM	The software data model component contains the language-specific scanners and parsers. The base class AnnotatedFile is used by text and binary files.	4,500	15
ZWIN	Portable graphics interface. A single class interface for windowing and drawing functions is provided. Two separate implementations of the interface exist, one for MS Windows and one for the X Window System.	11,000	30
UTIL	Various miscellaneous classes for data structures, file access, etc. It also contains an interface to some common operating-system–dependent routines.	3,300	12
Total		28,700	81

would build upon our advanced development work. Building a separate tool had a number of advantages: We could develop a data visualization tool apart from the other DEC FUSE tools. We could implement it in C++ and thus use some of the design from the SoftVis tool, if not the code. The impact on current tools was minimal: only small changes to their user interfaces and an added capability for sending data to the Data Visualizer were needed. By implementing a separate tool that receives messages from other tools, we would be following the style of tool integration used in the DEC FUSE environment.

Many changes had to be made to the prototype to move this work from advanced development into a product. Functions had to be added and removed. The design was changed in a number of places. Some changes resulted from the requirement to follow the tool integration standards for the DEC FUSE product. Other changes were merely good ideas that came about once we started the work of integration.

Data Visualizer Tool

Two major features were added to integrate the Data Visualizer tool into the DEC FUSE programming environment. First, all the data that composed the view was coming from outside the tool, unlike the prototype where data for the view was generated internally by analyzing source files. Now activities performed in other tools would generate this data and send it to the Data Visualizer. Second, multiple tools would be sending data that would need to be merged within the Data Visualizer into a single view. The remainder of this section summarizes the features in the Data Visualizer tool.

The Visualization DataSet File The Visualization DataSet file is used to pass information to the Data Visualizer for display. It contains two types of data. Software component data describes the files, directories, libraries, and functions to be visualized. Event data describes the data to be associated with these components. The types of events are defined in the file by the tool creating the file, but they must adhere to one of the predefined formats. An example of an event is a memory leak detected by a memory analysis tool. In the file, the memory analysis tool defines an event type for memory leaks and then passes as many events of this type as there are leaks detected. By allowing event types to be defined in the Visualization DataSet file, the Data Visualizer can easily support any tool that creates a file in this format.

Each set of events sent to the Data Visualizer from a particular tool is logically grouped into an entity called a DataSet. For example, a single DataSet contains all the results from a single search tool inquiry. Subsequent searches yield separate DataSets.

Condensed File Views In this paper, software components are shown in both the condensed file view introduced in Figure 2 and the 3-D view depicted in Figure 4. Each of these gives the tool a concise, information-dense representation capable of displaying up to 30,000 lines of source code. Program structure is revealed by the indentation and color coding.

Event Highlighting, Filtering, and Tracking Events in the DataSet are highlighted on the screen in a number of ways. Event types are assigned a color, and that color is used to color the line of the associated event. The coloring can occur in the foreground of the line or the background. Once a user's attention has been drawn to the line, the user can obtain more information about the event at that line from the small descriptive window that appears whenever a hot cursor is moved near that line. Figure 9 shows an example produced by the Data Visualizer tool. In addition, when the event contains more information than can be displayed on a single line, for example, when a complete program call stack is logged with the event, a separate window appears with this information. This is also shown in Figure 9.

The tool's legend/filter control window shown in Figure 10 serves the dual purposes of providing a color key to the events that appear in the view and a mechanism for toggling on/off the appearance of events of a particular type. This control window also allows the user to toggle on/off the appearance of all the events in a DataSet. When multiple DataSets are present, they are placed on top of each other. Each DataSet can be thought of as a transparency that contains only the event's highlighted coloring. These transparencies are stacked on top of each other (the user can control the ordering) to show all the events together.

The Data Visualizer also provides a mechanism for keeping track of events that are seen or unseen by the user. This feature can be used when there are many events to examine and the user needs assistance in tracking what work has been finished and what remains to be done. This information can be saved between invocations of the tool so that a user can put this work aside and come back to it at a later date.

Merging DataSets As mentioned earlier, one of the important features that was added was the ability to merge the data received from multiple tools into a single displayed view. This allows the combination of the results of two or more tools that normally could not be merged or even know of each other. For example, the output from a memory analysis tool that shows where memory leaks occur and their size can be combined with the output from a search tool that locates the occurrence of a function name in the program.

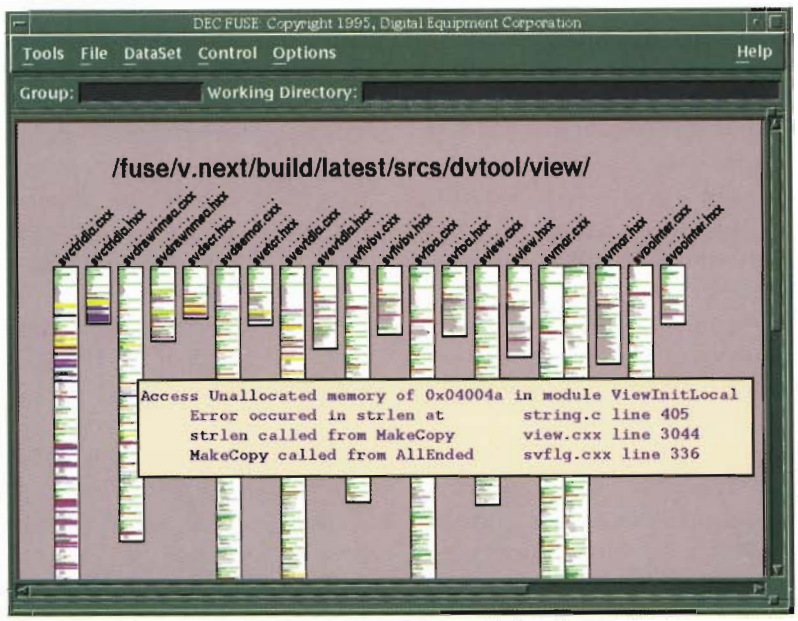


Figure 9
Highlighted Event with Call Stack

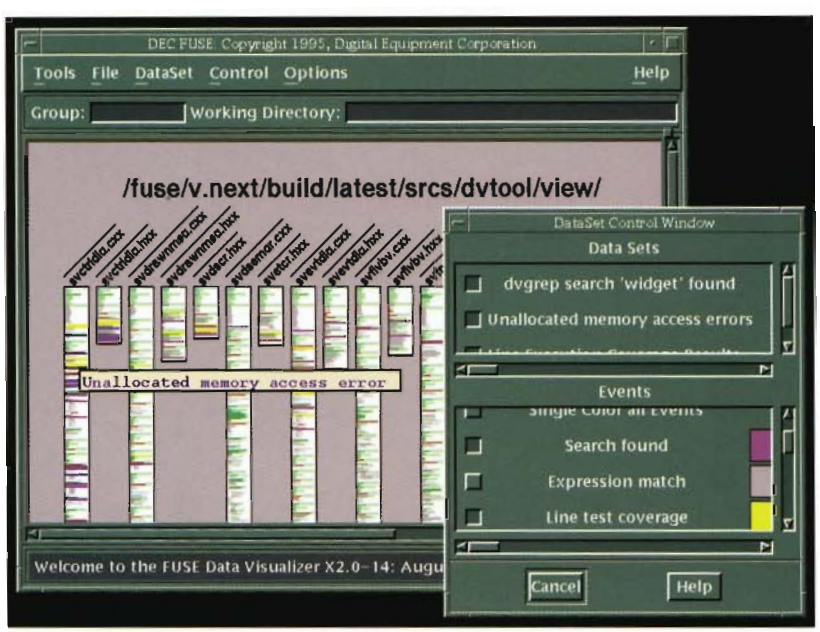


Figure 10
Event Filtering

The tool uses a number of methods for merging DataSets, and the type of merge that is performed depends on the types of events. The simple transparency model described earlier explains how events can be additively combined to display the sum of all events. In this model, when two or more events are associated with the same line in a file, they are treated as separate events that pertain to that line. For some event types, however, this is not the case. The tool supports the combination of same line events in different ways. For example, two runs of a performance analysis tool generate line execution times that can be combined by averaging the execution time values to give the user a reading on the average performance of the code. As an alternative, these same two events can be combined by creating a new event that shows the difference of the execution times to reveal improvements that may have occurred between runs.

Integration with Other DEC FUSE Tools The Data Visualizer is well integrated with the other tools in the DEC FUSE programming environment. The profiler, the heap analyzer, and the search tool all have the ability to send data to the Data Visualizer at a user's request. The Data Visualizer makes good use of the DEC FUSE editors to examine source code in detail. From within the Data Visualizer, the user can double-click at any point in any of the displayed files to have that source loaded into their preferred editor. This capability is shown in Figure 11, where the results obtained from the search tool are used to create a view in the Data Visualizer and load files into the editor.

Revised Design

As seen in Table 2, some of the prototype components were reused in the final product design. We changed the SDM component internally to handle more data, but we retained the basic design. We also retained the design of the UTIL component. Since portability between MS Windows and the X Window System was no longer a concern, we redesigned the ZWIN component into the WinDraw component. Due to this change, the size of this component decreased by 7,600 lines of code.

In addition to modifying components, we developed three new components. The FUSETool component handles the code common to all the DEC FUSE tools. It contains abstract base classes that can be used to derive new tools. The DVTool component contains the main program and the bulk of the user interface code. The View DataSet File (VDSF) component provides functions for reading and writing these files. It contains class libraries for C++ programs and C routines.

Note that this design maintains some of the plug-and-play characteristics of the earlier design. Although the tool component no longer exists, the VO (Visual Object) and the view components are present and provide extensibility for future objects and views.

Conclusions

The last section gives an overview of the software design from advanced development into final product. The section concludes with some future plans for this work.

Project History

During the process of transferring this work from advanced development into a product, many important features were added to enhance the usefulness of this technology. The final product retained the ability to visualize large amounts of data in a condensed yet comprehensible format; it also included features, like event tracking and DataSet merging, that made it a much more useful productivity tool. Figure 12 shows how the design evolved over time. The events

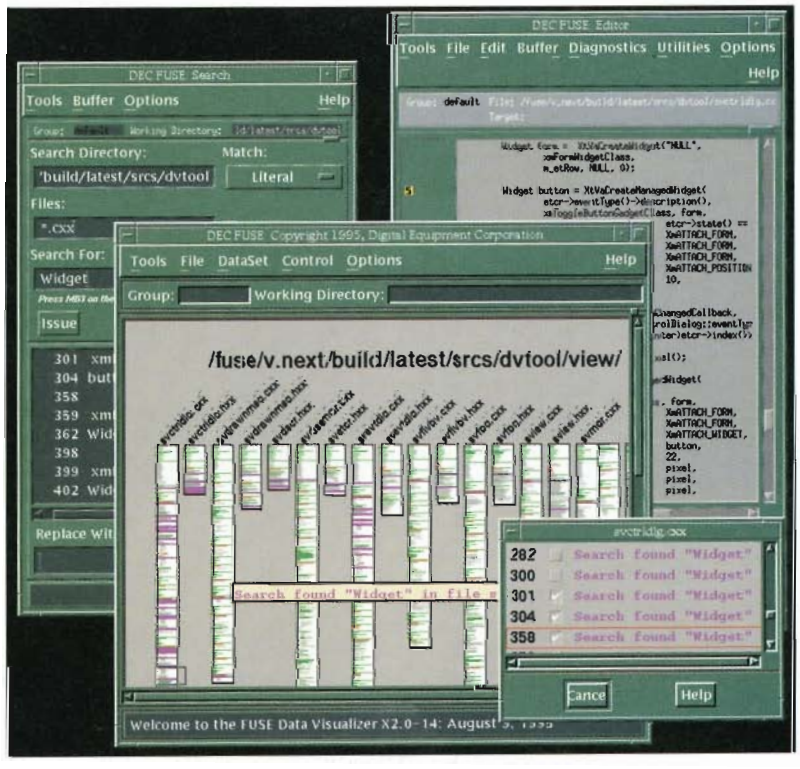


Figure 11
Integration with Other DEC FUSE Tools

Table 2
Components in the Data Visualizer

Component	Description	Lines of Code	Classes
FUSETool	Base class for building a DEC FUSE tool. Contains code common to all DEC FUSE tools.	3,000	8
DVTool	The Data Visualizer main classes. Contains the main program and most user interface classes.	2,400	10
VO	Contains the svObject base class and its derivations, the svFile, the svDirectory, and the svLibrary.	2,000	5
VIEW	Contains the svView class and its derivations, the svFileView and svFile3dView classes.	3,500	8
SDM	Software data model component. Contains the language-specific scanners and parsers. Defines the program's internal data model.	3,500	15
WinDraw	Provides C++ encapsulation of graphics drawing functions.	4,100	12
VDSF	The VisualizationDataSet Format component provides reading and writing routines for this file format.	1,000	4
UTIL	Various miscellaneous classes for data structures, file access, etc. It also contains an interface to some common operating-system–dependent routines.	2,000	8
Total		21,500	70

described in this paper occurred over the course of two years and three months. The advanced development project began in January 1993, and the final design of the Data Visualizer tool was complete in March 1995.

In Figure 12, the rectangles represent software components of the design. A software component is a collection of C++ classes that was designed to accomplish a single function; these components correspond to the design components described earlier in this paper. The oval shapes represent prototypes that were built from these components. Solid arcs connecting components with prototypes show which components were used to build that piece of software. Dotted lines between components show how components evolved over time.

Figure 12 indicates that the work involving 3-D objects and some of the early prototype components were never used. It also shows that the condensed file view component and the ZWIN component did evolve into the final product. Figure 12 further reveals that toward the end of 1994 several documents were produced, but no work was done on the design or any of the components. During this period of negotiation and redesign, the advanced development technology was being converted into a product.

Future Work

We would like to expand the capabilities of the Data Visualizer tool in several areas.

Many of the capabilities for merging DataSets are not available for selection by the user. We would like to extend the tool to have the added flexibility of allowing the user to decide how DataSets should be merged and how events should be combined. For example, the tool might show only the intersection of two DataSets, that is, display only those events that point to a file-line combination that is common in both sets.

We will also consider other ways of displaying in a condensed file format and additional types of files to visualize. The file types might be complete directories shown as a single, condensed object, or shared and nonshared libraries as a single object.

We have an ongoing effort to take the output from existing tools and visualize it in this tool.

Final Remarks

The decision to include the Data Visualizer tool in the next major release of the DEC FUSE programming environment was not an easy one to make. Many important features were being considered, but not enough resources were available to perform the work. Prioritized goals were established, and all work items were evaluated against these goals. The Data Visualizer tool was included for two important reasons. First, it supported the short-term goals of the project by adding features that current tools could use in the upcoming release. Second, it provided long-term benefits by opening up the DEC FUSE product to new capabilities in the area of software visualization. We believe that the presence of both these reasons was necessary for its inclusion in the DEC FUSE product. Had it provided support for only the short-term product goals, it would have been evaluated against the many other short-term work proposals and probably would not have been selected. Had it supported only the long-term goals, it would have been left out for lack of ties to the current tools.

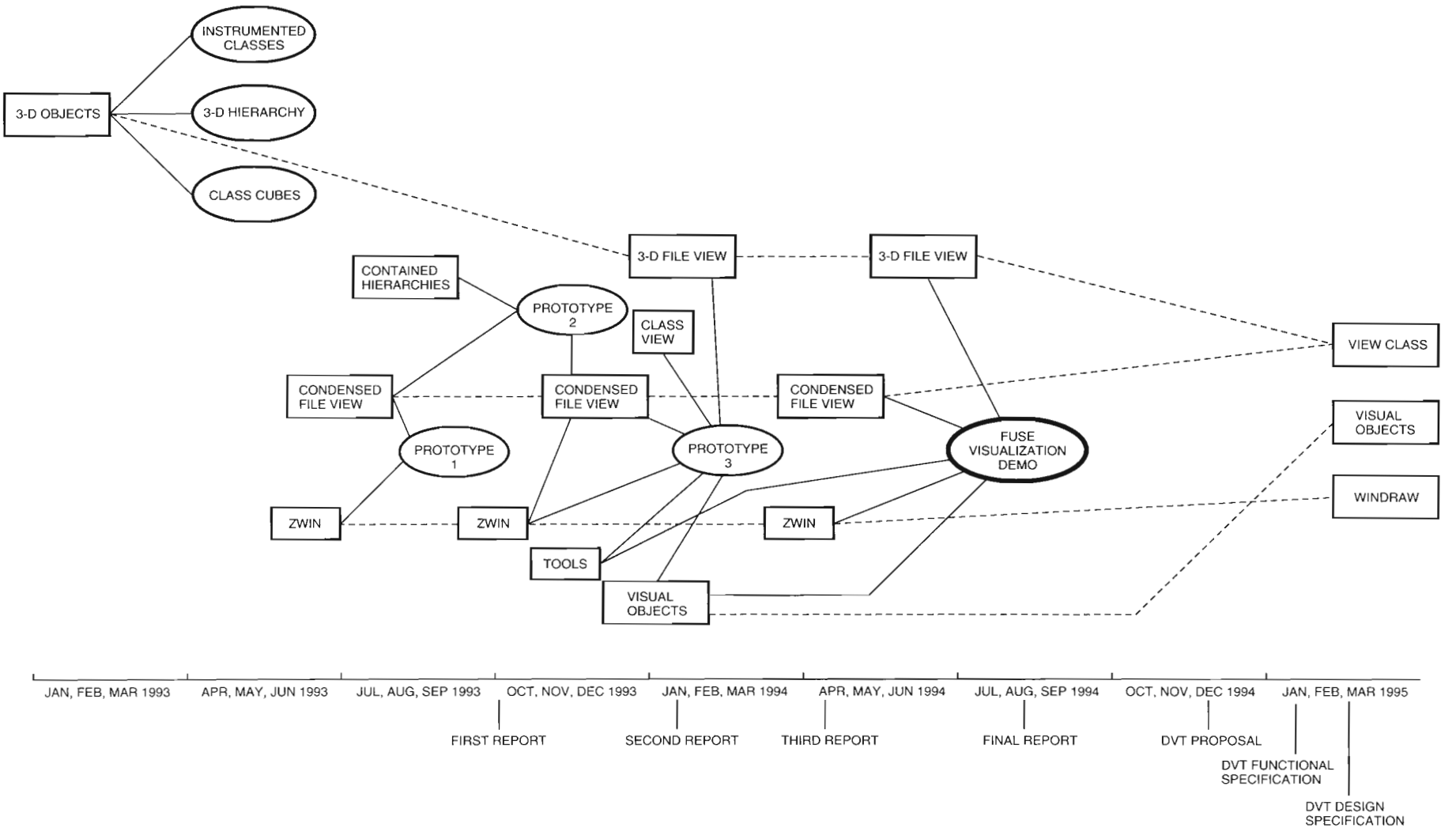


Figure 12
Project History

Acknowledgments

I would like to thank a number of people who supported me during this effort: John Ellenberger for his continuing guidance throughout the entire process; Mike Candella for the early work we did together in the Advanced Development Group; Glenn Lupton for his help in deciding how this prototype would fit into the DEC FUSE product; the DEC FUSE management team for supporting and encouraging this work; and finally, everyone on the DEC FUSE development team.

References

1. R. Hart and G. Lupton, "DEC FUSE: Building a Graphical Software Development Environment from UNIX Tools," *Digital Technical Journal*, vol. 7, no. 2 (1995, this issue): 5–19.
2. S. Eick, "SeeSoft—A Tool for Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11 (1992): 957–968.
3. S. Eick, "Graphically Displaying Text," *Journal of Computational and Graphical Statistics*, vol. 3, no. 2 (1994): 127–142.
4. S. Eick, M. Nelson, and J. Schmidt, "Graphical Analysis of Computer Log Files," *Communications of the ACM*, vol. 27, no. 12 (1994): 50–56.
5. K. Perlin and D. Fox, "PAD—An Alternative Approach to the Computer Interface," *SIGGRAPH 93 Proceedings* (1993): 57–64.

Biography

Donald A. Zaremba
The project leader of the FUSE Data Visualization team, Don Zaremba is a principal software engineer in Digital's Unix Development Environment Group. He was responsible for designing and implementing the Data Visualizer tool. Since joining Digital in 1980, Don has contributed to the DEC Test Manager project and has worked on software development tools and fault analysis tools. He received a B.A. in mathematics from the State University of New York and an M.S. in software engineering from Wang Institute.

Eric A. Newcomer

Multivendor Integration Architecture: Standards, Compliance Testing, and Applications

The Multivendor Integration Architecture (MIA) is a user-driven initiative that addresses the practical application of open systems software standards to business requirements. This paper provides historical background and context for this standardization effort and describes Digital's contributions to the effort, particularly in the area of distributed transaction processing. Digital complied with the MIA specifications, integrated compliant products into a complete platform, and delivered a large application on the platform.

In today's competitive environment, an enterprise's computer systems help determine its success or failure. The need for large enterprises to separately manage applications on different computer vendors' platforms distracts the enterprises from performing their main business functions and adds to their operations cost. Corporate mergers and acquisitions often compound the problem.

While the business need for high-quality computer systems has never been greater, established computer users find themselves in a poor position due to the tremendous burden of their legacy systems. Newer companies almost automatically gain a competitive advantage from their more flexible, state-of-the-art computer systems.

The availability of open, standards-based systems enables critical business systems to be built on a common platform that can be purchased from multiple vendors at competitive prices. This offers everyone the same level of basic functionality with which to build new systems. These systems must be capable of integrating components from multiple vendors into a single, large application.

This paper provides background information for user-driven standardization efforts, with a focus on Nippon Telegraph and Telephone's (NTT's) Multivendor Integration Architecture (MIA). The paper discusses the MIA's principles, including three multivendor interfaces, NTT's major types of computer processing, specification development, and Digital's approach to addressing integration problems related to transaction processing (TP). Also discussed are implementation and systems integration issues and the delivery process. Digital's contributions to the open systems software integration effort are described. Digital was instrumental in defining the MIA specifications for TP, and it developed the first MIA-compliant application.

User-driven Standardization Efforts

About 25 years ago, NTT, one of the world's largest corporations, developed its first computing system procurement specifications. These detailed specifications

included designs for special hardware and operating systems to meet the enterprise's demanding requirements.

The procurement specifications focused on systems of sufficient capacity and robustness with which to automate the fundamental business operations of a large telephone company. They did not require portability or interoperability. NTT presented the specifications to Hitachi, Fujitsu, and NEC and ordered hardware and software that conformed. In addition to the Japanese suppliers, IBM also responded to the procurement request and became an NTT supplier.

Following the successful implementation of the original specifications, NTT developed applications on top of the various vendors' platforms. Like many other large enterprises, NTT created separate teams to tackle the vendors' systems individually.

In 1988, NTT established the MIA consortium to resolve the inefficient practice of having separate teams develop and manage applications on different vendors' platforms. The consortium was charged with addressing the associated problems that interfere with the way these applications communicate, share code, share data, or move to a new technology base.

The MIA initiative was conducted as a Japanese industrial collaborative research project with the goal of resolving the problems of multivendor application environments. NTT invited computer vendors to join the project by issuing a public subscription announcement and then selected participants from among the respondents. Fujitsu, Hitachi, NEC, and IBM were the first consortium members. Digital was also selected because of its expertise in networking and client-server computing. The MIA initiative set out not only to resolve the problems with a multivendor environment but also to move NTT's computing systems forward by incorporating distributed processing functionality.

One of NTT's goals was to eliminate all visible differences among the vendors' platforms. "Visible" meant perceptible to (1) the humans who interact with the computers as end users, in application development and deployment, in system administration, and in network configuration and management, and (2) the protocols for communication between the different vendors' computers. A guiding principle of the MIA initiative was that the systems with which people interact should appear identical, regardless of the manufacturer who created the hardware or software being used or the purpose for which it was being used.

As a member of the MIA consortium, Digital helped develop detailed specifications that met NTT's requirements for open systems software components that any vendor could implement. In particular, Digital developed new multivendor specifications for distributed TP, an area of computing for which standards did not exist.

The results of the MIA project were published in 1991 as 11 volumes of detailed procurement specifications that describe a complete application development platform for large-scale systems.[1] Applications created using software that conforms to the specifications can be developed and implemented on any vendor's computer.

The concepts behind the MIA specifications were put to the test at a public demonstration at Interop Tokyo in July 1994. After considerable debugging and testing, the concepts were proven to work.[2] The next measure of success is whether sufficient demand and cost savings exist to induce vendors to market conforming products, in particular, off-the-shelf products.

Digital's involvement in specifying solutions to user-driven open systems software requirements continues at the Service Providers' Integrated Requirements for Information Technology (SPIRIT) consortium, which is sponsored by the Network Management Forum. SPIRIT members include the world's largest telecommunications service providers and computer vendors. The MIA specifications were submitted as base input documents for SPIRIT, along with other documents from AT&T, Bellcore, BT, and ETIS (a consortium that represents 27 European postal, telegraph, and telephone administrations).[3]

It is unknown whether this user-driven approach to standardization will succeed and meet the important goals of portability, interoperability, and multivendor procurement.[4] Nonetheless, users and vendors are learning some important lessons as a result of the users' strong efforts in this area.

MIA Principles

When NTT turned its attention toward creating the MIA procurement standards, it began to attack the problem of multivendorization, which NTT believes is strategic to its future business. "Because a computer system must be able to provide as broad a range of business services as possible, it is desirable to construct such a computer system flexibly enough to include different computers, each of which covers the area of business in which the vendor's model is the most powerful."[5]

Early in the MIA project, NTT established the basic requirement that solutions be based on open systems standards where possible. However, since the corporation's existing complex legacy of applications was critical to business operations, the new standards had to allow for the same degree of functionality and robustness as the software for the existing platforms. Also, if it was to replace its current applications with applications that took advantage of commodity technology, NTT needed a way to migrate to the new while interoperating with the old. "Based on the assumption that

a variety of hardware and operating systems of vendor-specific design is widely accepted in the general-purpose computer market, MIA specifications must be a feasible extension of, and coexist with, vendor-specific architectures."[5]

The MIA effectively grouped related functionality to match the existing requirements for business applications and added support for distributed client-server computing. Using the resulting architectural framework, the MIA consortium matched existing standards to NTT's needs, identified missing functionality, and created new multivendor specifications to achieve the additional functionality.

Three Interfaces

At the start of the MIA project, NTT identified what it considered the three most important issues of multivendorization:

1. Duplicated development of application programs
2. Difficulties in resource sharing
3. Differences in operating methods[5]

For each of these problems, NTT identified solutions in terms of standard, i.e., multivendor, interfaces, as follows:

- Application portability using standard application programming interfaces
- Interoperability using standard communication protocols
- Common user interface using a windowing style guide

Figure 1 illustrates the basic architecture as specified by the MIA consortium. The configuration incorporates three systems—the end user, the departmental computer, and the host computer—and includes three types of interfaces—human user interface (HUI), application programming interface (API), and systems interconnection interface (SII). The figure represents the fundamental goal of MIA conformance for each vendor, i.e., to offer conforming interfaces and protocols that allow NTT to purchase the same level of compatible software functionality from multiple vendors and create new applications that are inherently distributable, portable, and interoperable. Another reason NTT focused on these three interfaces was that if the MIA specifications contained too many low-level interfaces, the vendor-specific strengths would be removed and the specifications would not support the NTT strategy of multivendorization.

Through the standardization of the three interfaces, NTT anticipated that an end user would be able to use any display device without knowing the vendor (via the HUI), a programmer would be able to write a program that would run equally well on all platforms (via the API), and a computer from one vendor could be connected to a computer from any other vendor using common systems interconnection protocols (via the SII).

Additional types of interfaces and protocols that were outside the scope of the MIA specifications are being addressed by the SPIRIT consortium. For example, SPIRIT has taken on the task of standardizing the system management interfaces and protocols. At the start of the MIA initiative, NTT decided that the best use of time and resources would be to standardize the HUI, the API, and the SII.

Major Types of Computer Processing

NTT categorized its computing activity into four types: real-time processing, transaction processing, interactive processing, and batch processing. Figure 2 illustrates the processing types and interfaces addressed by the MIA specifications. Note that the specifications did not address real-time processing issues.

NTT included the area of TP because the company had a huge investment in developing and running TP systems and because its business relied on TP systems such as billing, inventory control, and directory assistance. The opportunity for return on investment was therefore high for this critical application area. Data

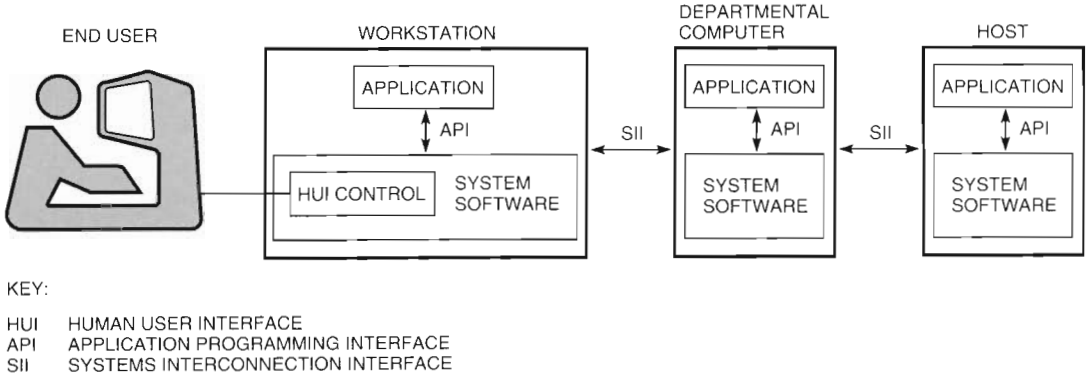


Figure 1
MIA System Configuration

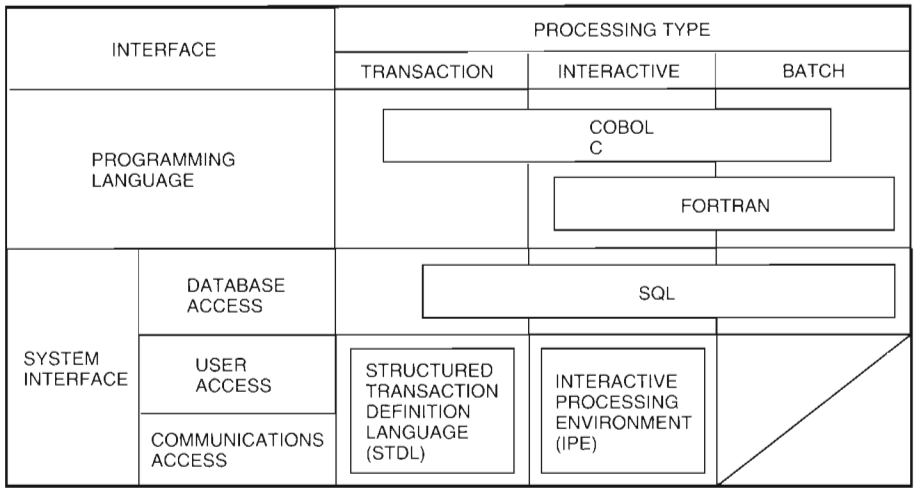


Figure 2
MIA Processing Types and Interfaces

integrity, remote access, and system reliability are the key characteristics of TP that needed to be supported through standards compliance to fully realize the cost savings potential of the MIA.

In the area of TP, no international standards existed for the two most significant interface areas NTT had identified as candidates for multivendorization: the API and the SII. This deficiency created one of the biggest problems that the MIA consortium had to resolve and later gave rise to a large systems integration and application delivery challenge with respect to the MIA.

NTT required the MIA TP specifications to support remote, distributed transactions. MIA TP comprised specifications for multiple programming languages and network protocols and therefore became the widest integration point that had to be achieved.

Developing the Specifications

As the first step in specifying solutions to the problems that it put forth to the MIA consortium, NTT produced user requirements. The user requirements evolved over the course of the project as new questions arose that needed clarification from NTT's business sector. Meeting user requirements was the final verification of the specification output at the end of the project. In addition, the consortium had to develop specifications that could be implemented by any vendor.

For the area of TP, NTT asked each vendor in the MIA consortium to submit a proposal for a new multivendor specification and selected Digital's Application Control and Management System (ACMS) TP monitor proposal as the base on which to build.[6] A TP monitor is a software component that provides functions required for TP applications, such as transaction coordination, display management, and performance improvements.

NTT selected the ACMS proposal as the base of the new multivendor standard for two reasons: the ACMS TP monitor included a high-level TP control language called the Task Definition Language (TDL), which could be made portable more easily than a lower level API, and the monitor used a remote procedure call (RPC) communications model, which is easier to program than a peer-to-peer communications model. That is, the ACMS technology was determined to provide the best solution to NTT's requirements for multivendor portability and distributed processing.

The problems to be resolved by the consortium vendors, consistent with the principles of multivendorization set by NTT, were

- Portability
- Interoperability
- Common user access

Historically, portability has best been achieved among vendor platforms by using a high-level language such as C or COBOL. This principle was true for the MIA, except that the MIA consortium found it necessary to produce profiles of programming language standards. The C and COBOL standards are not sufficient to achieve portability because so many of the specification rules are subject to a variety of interpretations among vendors, and architectural language limits are not defined.[7,8]

An MIA profile of a programming language standard references the standard specification and modifies it to improve portability. In the case of the MIA COBOL profile, national text support is mandatory for portability of international language features. The X/Open Company adopted this work as the basis for their COBOL national language support and accordingly published the X/Open COBOL specification.[9]

The MIA COBOL profile also deletes sections of the ANSI COBOL specification that contain optional syntax that a vendor may choose to implement. Finally, the MIA COBOL profile sets common language limits such as the maximum length of a text string and the number of parameters supported on a procedure call. The resulting profile allows programmers to create source programs that are portable to any vendor who conforms to the MIA specifications.

The MIA programming language profiles were required because of the way vendor-driven standards are typically written. The goal of vendor-driven specifications work is to allow the widest possible interpretation of architecturally significant issues such as integer precision, file system naming rules, and memory manipulation, and thereby to allow the widest possible implementation and adoption.

The MIA C profile adds rules for defining the conversion of a signed integer into an integer of smaller or equal size and for defining the results of dividing by a negative integer. Neither of these semantics is defined in the ANSI specification because they tend to vary according to vendor architecture. The MIA C profile also defines wide-character handling in the print and file manipulation functions so that programs supporting international language character sets would be portable.

Efforts to address these portability issues, such as the X/Open XPG portability specifications, usually describe or catalogue the problems so that the programmer can avoid them.[10] MIA places the burden of ensuring application source code portability on the vendor instead of on the programmer.

No language standard existed for the MIA processing area of TP, however. Although some protocols existed for various degrees of interoperability, none existed for complete distributed transaction coordination.

Solving the TP Problem

Perhaps the most significant aspect of the MIA effort is its approach to resolving problems associated with distributed TP. Typically, TP applications are very large and involve strict requirements for performance and availability. TP applications implement the daily operations of a business. Some of the better-known examples include travel reservation systems and automatic teller machines. The term "transaction" is derived from the term "business transaction," which means an exchange of goods or money between two individuals or businesses, or some combination thereof.

Transactions, when automated, take on additional properties because computer systems are subject to failure in ways that manual systems are not. Computer systems are electrical, and electrical failures can damage data storage media. Computer systems are networked, and communication failures can interrupt the completion of a business transaction such as a travel reservation that requires the participation of multiple computers at multiple sites.

A computer transaction uses logging to ensure that business data is captured reliably or not at all. Perhaps most important, a computer transaction ensures that business computer systems recover quickly from any type of failure and begin processing data again without manual intervention.

Because of the highly demanding nature of TP, vendor implementations of TP system software depend on the features of specific hardware and operating system architectures for the purposes of performance optimization and fast recovery. The mechanisms for accomplishing fast recovery are complex and difficult to implement on a multiple-user system. Although business data is shared, operations on the data must be isolated so that one operation does not overwrite the effects of another operation. When two simultaneous requests arrive to update the same bank account, for example, the ending balance may be incorrect if the two updates are not properly serialized. Such errors can occur unless transactions are used to isolate and serialize the updates. Failures of media or communications can result in inconsistent data.[11]

These difficulties and others have deterred standards bodies from addressing the area of TP. Consequently, the market is dominated by proprietary solutions. Users are liable to be locked in to a particular vendor and to have difficulty achieving the benefits of competition.

The MIA TP specifications were designed to address these problems and to counter the shortcomings of the traditional vendor-driven software standardization process. MIA TP eliminates vendor-specific differences by adding a high-level language layer on top of proprietary TP monitors and by adding a common protocol at the lower layers for interoperation.[11] The only restriction that MIA places on the underlying software or platform is that it must be sufficient for implementing the specified TP functionality. Otherwise, vendor and user investment in existing systems is preserved.

The MIA consortium based the MIA TP protocol standard on the International Standards Organization/Open Systems Interconnection (ISO/OSI) TP protocol, and on the Open Software Foundation's (OSF's) Distributed Computing Environment (DCE) RPC, both of which were newly released.[12] To balance the risk of adopting a new technology, the MIA consortium chose IBM's Systems Network Architecture (SNA) Logical Unit 6.2 (LU 6.2) as a short-term alternative solution.

The MIA transactional communication specification combined DCE RPC as the data transport and OSI TP for the two-phase commit protocol. The resulting protocol was called the Remote Task Invocation (RTI)

protocol, which was subsequently adopted by X/Open as the basis of their TxRPC specification.[13,14] Figure 3 shows the resulting MIA TP model.

To solve the portability problem, the consortium began with Digital's proposal based on the ACMS TP monitor's TDL and developed a new Structured Task Definition Language (STDL), which is a modular, block-structured language very similar to TDL.[15] The consortium eliminated vendor-specific syntax, ensured that STDL's features met NTT's user requirements, and conducted implementation studies to verify that the new language could be implemented on top of each vendor's existing proprietary TP monitors.[16] Figure 4 illustrates the layering of the new MIA TP language on the MIA TP protocol.

Because the MIA was based on standards as much as possible, the MIA TP work also had to be largely based on standards. Therefore, the STDL specification was integrated with the standard languages C, COBOL, and SQL to provide complete, portable application functionality.[17] The consortium mapped the data types among the four languages and specified interlanguage call semantics.

STDL procedures can call and be called by C and COBOL procedures. STDL implements the TP-specific functionality that standard C and COBOL lack. Examples of this functionality are beginning and ending a transaction, handling transaction exceptions, automatically restarting transactions, and coordinating multiple transactional resource managers (i.e., databases, files, and queues) locally or across remote TP systems in a network.

Adopting STDL as a new language represented a practical way to add TP-specific functionality in a multivendor environment while allowing the C, COBOL, and SQL languages to be used as specified in international standards. This approach did, however, result in additional integration problems. It was necessary to ensure that STDL procedures worked with C and COBOL procedures as well as with SQL and within the entire TP environment, which encompassed a large part of a platform's capabilities. An additional

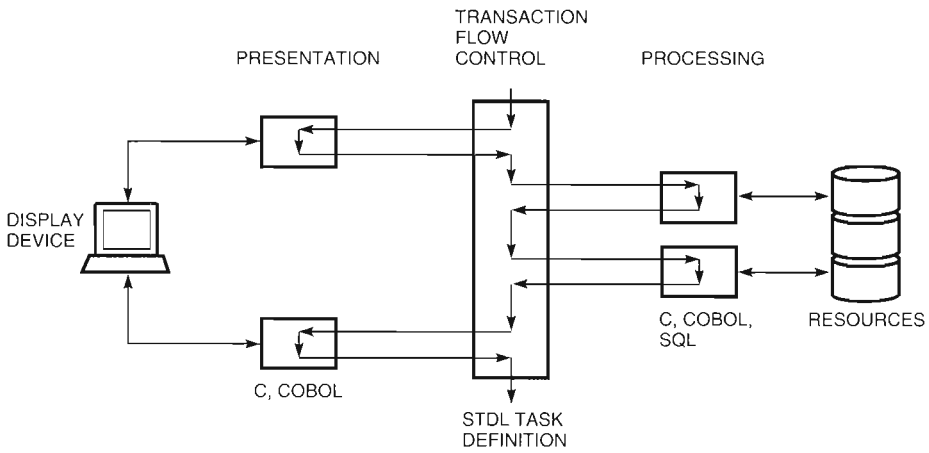


Figure 3
MIA Transaction Processing Model

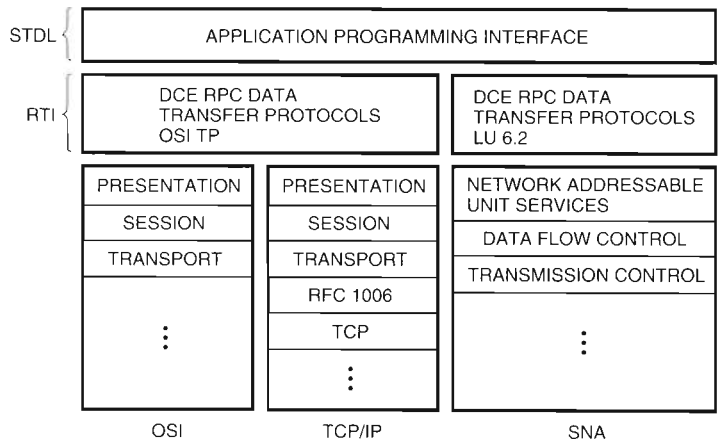


Figure 4
MIA Transaction Processing API and Protocol

benefit results from the use of a compiler to check STDL syntax and semantics, thus reducing the instance of execution errors.

Implementing the MIA Specifications

Because the architecture was defined at the interface level, the implementation and system integration problem for vendors entailed identifying the components with conforming interfaces and assembling them on the platform that met the MIA specifications. Although focusing on three interfaces was practical with respect to completing the 11 volumes of the MIA specifications in approximately 18 months, such a scope left uncovered many areas of technology that the vendors intending to implement MIA would have to provide for themselves. System and network management, computer-aided software engineering (CASE) tools, and testing and debugging tools are examples of items that would have to be integrated with the components that complied with the specifications.

Table 1 lists the primary areas of the MIA specifications and the types of standards included in each area.[7,8,12,14,15,17–24]

The MIA specifications' practical approach to resolving the problems of portability and interoperability include carefully documenting where the vendor differences continued to exist among the implementations of the standards. "In general, the amount of information transferable between development and execution environments under the original MIA procurement specifications is less than that transferable when both environments are provided by the same vendor."[1] Some vendor-specific coding, for example, including file names in source code programs, could not be standardized by MIA because of fundamental vendor differences. Instances of such unresolvable problems were carefully documented.

The amount of portability gained by following the MIA specifications was significant, however, as compared to the amount that would be gained without using the specifications. The following example of defining the integer size illustrates the benefit derived from having the MIA C specification.

A C program written using a vendor's compiler that interprets a long integer data type as having 16 bits will not work correctly when ported to another vendor's compiler that interprets the same data type as having 32 bits (which is an acceptable interpretation according to the ANSI/ISO C specification). Typical solutions to this problem have been to document the problem and instruct programmers to recode when porting their programs, or to have programmers write their original programs so as to avoid the problem.

The MIA C specification resolved this problem and similar problems in that it represents agreement among the MIA consortium vendors on a common interpretation of the ANSI/ISO C specification. Because the MIA specifications are procurement specifications, vendors must conform to the MIA C specification when responding to MIA-compliant requests for procurement (RFPs) from NTT.

Table 1
Areas of MIA Specifications and Associated Standards

Areas of MIA Specifications	Standards
API	
COBOL	ISO 1989:1985, ANSI X3.23-1985
FORTRAN	ISO/IEC 1539-1991, ANSI X3.198-1992
C	ANSI/ISO 9899
STDL	MIA specification adopted by SPIRIT and submitted to X/Open
SQL	ISO 9075-1:1992
HUI	
OSF/Motif	OSF/Motif Style Guide, Release 1.2
IBM's Common User Access	No standard established
OPEN LOOK	No standard established
SII	
MIA TP protocol	MIA RTI specification adopted by X/Open as the TxRPC specification
OSI TP	ISO/IEC 10026-1:1992
MHS X.400	ISO/IEC 10021-1:1990, CCITT X.400-89
FTAM	ISO 8571-1:1988
TCP/IP, FTP, SMTP, TELNET, SNMP, UDP, CMIP	Internet protocol suite
X.25	ISO/IEC 8208:1990, CCITT X.25-89
ISDN	CCITT I Series
Ethernet	ISO/IEC 8802-3:1993, IEEE 802.3-93

Implications for Systems Integration and Application Delivery

NTT awarded Digital the first contract to deliver an MIA-compliant application. NTT selected its List Maintenance System (LMS), the application that manages the telephone number database used to produce telephone directories for all of Japan.[2] One purpose of the LMS was to sufficiently test the specifications. The LMS procurement involved 60 software products from a variety of Digital engineering groups. The components had to be modified to meet the specifications and then integrated, tested, characterized, and delivered on the OpenVMS operating system. The target configuration of three VAX 10000-630 systems in a VAXcluster configuration supported more than 10 client sites throughout Japan. The contract includes software, hardware, and services. Figure 5 illustrates the LMS application.

Of the 60 software components in the LMS platform delivery, 27 were required for conformance to the MIA specifications. Although the remaining 33 components addressed application areas outside the scope of the MIA specifications, these products had to be integrated with the MIA-compliant products, tested, characterized, and verified, thus making the integration effort more complicated.

Even though NTT realized some benefits from the standardized products that it procured according to the MIA specifications, it faced a dual systems integration problem. Delivery required complying with the specifications and also complying with the detailed terms of the specific RFP for the LMS.

Figure 6 illustrates the system verification and characterization process carried out by Digital's Systems Application Integration and Engineering (SAIE) group. This was the key effort in responding to the MIA-based procurement request.

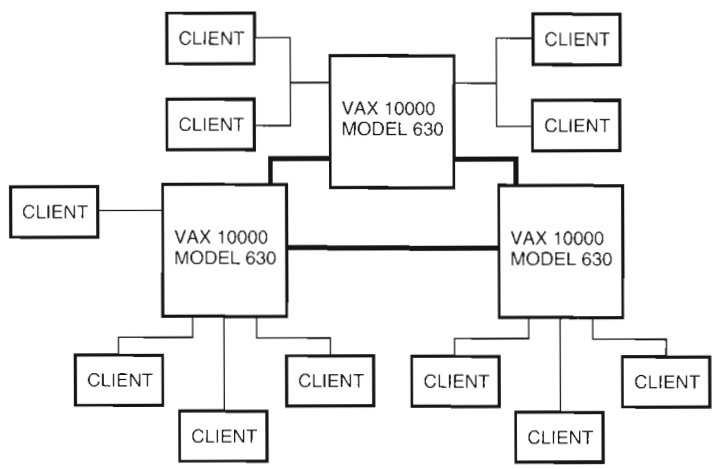
Digital established a special-purpose production systems program office (PSPO) to oversee the entire process of delivering the MIA-compliant RFP. This program office was modeled after the successful Alpha program office.[25]

A production systems board of directors represented the various engineering departments whose component products were included in the LMS. The board's function was to resolve priority and budget conflicts among the various departments. This group met monthly.

A special project forum was established with representatives of the individual products and engineers who could resolve technical problems and fix bugs that surfaced in the integration and testing activities. This group met weekly.

The SAIE group provided a "sandbox" for component product groups to install and test their products on the specific version of the OpenVMS operating system on which the components were to be delivered. This process was repeated for operating system upgrades and was made more difficult because initially a special version of the OpenVMS system was required to fully meet the terms of the RFP, in particular, to provide Japanese language support.

After the components were installed in the OpenVMS operating system, SAIE engineers verified that the components worked together by running test



SPECIFICATIONS:
3 VAX 10000-630 SYSTEMS IN A CLUSTER
11 CLIENT SITES
60 SOFTWARE COMPONENTS
STDL TP MONITOR
500-GB DATA REQUIREMENTS
MIA-COMPLIANT PLATFORM

Figure 5
List Maintenance System

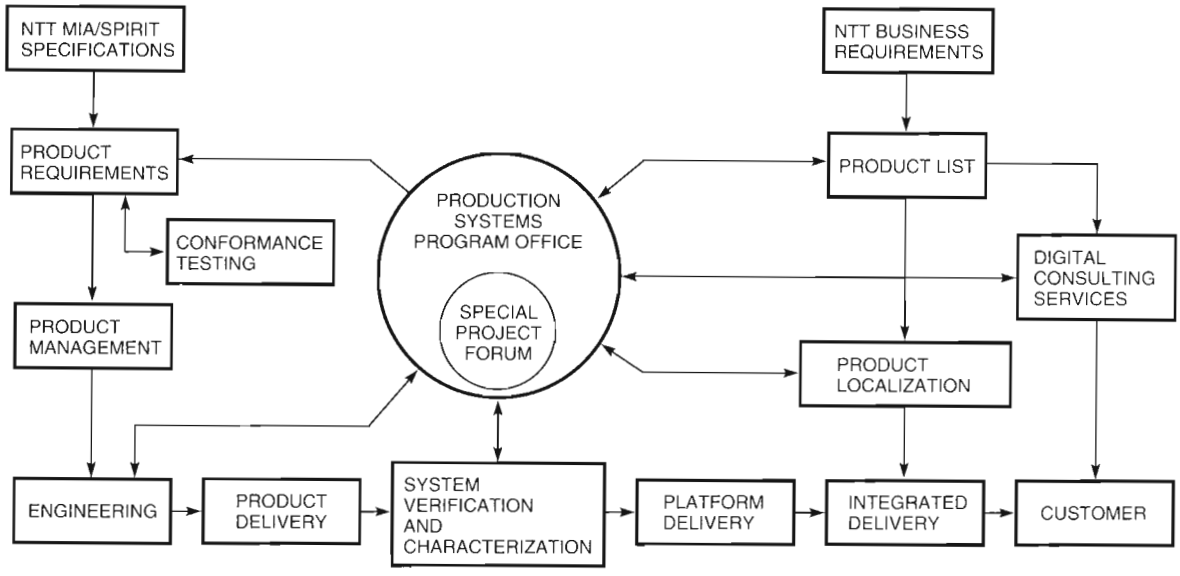


Figure 6
Digital's MIA Systems Integration Process

applications and characterized the overall performance of the platform as configured. Any problems that arose during this testing and characterization work were routed back to the component product groups by means of the special project forum. Finally, the program office coordinated the delivery to the local Digital office in Japan and to the customer (NTT).

The integration effort for the LMS uncovered more than 170 bugs, of which 25 were major obstacles. If Digital had not undertaken the integration effort, the problems would have shown up at the customer site and jeopardized the contract. Of the bugs, nearly 50 percent were directly related to integrating the various components on the common platform.

For example, one bug involved a fatal clash between versions of a threading package. Two LMS component products had incorporated incompatible versions of the same threading package without considering the potential problems that might arise if the two separately developed components were integrated and tested on the same platform.

Another problem resulted from the upgrade from the VAX C language compiler to the DEC C compiler, which was to comply with the new ANSI standard for the C language. While upgrading its C compiler to comply with the ANSI C standard, Digital altered the semantics of the associated run-time library. Most new software components are coded using C, so nearly every component on the platform was impacted.

During the 18-month period that the program office, the board of directors, and the project forum supported the LMS effort, 56 releases and patches were provided for LMS integrated products. Each time a new version of the operating system or a major component was released, the integration, testing, and characterization process had to be repeated.

The major lesson derived from the experience with MIA was the type of project and program management required to deliver a complete platform for enterprise-level computing on a large scale. Additionally, Digital engineers learned to work with other vendors to ensure the compatibility of Digital's implementation of the MIA specifications with the other vendors' implementations.

Digital remains very interested in pursuing opportunities to resolve enterprise-wide computing platforms for its large customers. The most significant problem to be solved is the systems integration problem. The MIA effort proves that products from different engineering groups within Digital need to be installed, tested, verified, and characterized before being delivered to the customer for use in a large application. Systems integrators can anticipate that the integration problems discovered during the LMS project will be compounded in an effort that involves software components from multiple vendors.

Large enterprise-level applications such as the LMS cannot be mass produced. The number of these large applications is small, and the needs of individual enterprises can vary significantly, even within a single industry segment such as telecommunications. Digital's experience with the SPIRIT consortium follow-on to MIA has demonstrated this.

It is therefore important to preserve the learnings about how the MIA platform was put together and, of lesser importance, to be able to exactly replicate the

platform delivered to NTT for the LMS. Digital needs to be able to work with large customers such as NTT in the future and to complete large projects such as the LMS, backed by an internal systems integration and delivery organization.

Indeed, the systems integration problem grows more complex in a world in which products from multiple vendors are routinely required to work together in providing the solution to a large application's requirements. Customers tend to look more and more toward contracting for the technical expertise needed to solve these problems.

Delivery

Delivering an MIA-compliant business solution involves several levels of integration, each with its associated problems. The first level is integrating the required functionality in specifications developed by independent standards bodies. The next is combining standards-compliant component products on a single operating system and hardware platform, while preserving the required interfaces and behaviors. Third is incorporating the additional products and features necessary to develop a specific application on the standards-compliant platform. Fourth is ensuring that compliant platforms from multiple vendors can work together. The integrated product set must then pass conformance testing and verification. When application development begins, additional integration issues arise that affect the overall process.

During Digital's implementation of the MIA specifications and the subsequent integration activity to combine the components on one platform, several problems were discovered in the specifications. These problems were reported to NTT and directed to one of the specification working groups, which had continued under the auspices of the consortium for this purpose. For example, after testing interoperability using the RTI protocol, the mapping of communication errors to STDL exception codes was found to be incorrect.

Ultimately, not all the goals of the MIA initiative were met. During the implementation and delivery effort, it became apparent that specifying a standardized HUI would not be possible. The use of a windowing system with a common look and feel and common principles of operation (e.g., a mouse, icons, and pull-down menus) was sufficient for end users, and the industry players were too widely split to endorse a common solution. Specifying a standard for the size and shape of an icon or for how to entitle entries on a pull-down menu became unnecessary as windowing systems converged on common design principles of operation.

STDL Maintenance and Conformance

Because STDL was a newly specified language, it required considerable maintenance. NTT carefully monitored the vendor implementations of STDL to ensure that all the MIA vendors interpreted the specification in the same way. NTT procured several STDL-based applications from different vendors. Consequently, vendors were able to experience the inevitable implementation problems in realistic situations. If NTT determined that a problem was or might be related to the specification, it encouraged the vendor to submit a problem report to the appropriate MIA consortium working group.

NTT defined conformance testing for MIA, including STDL. Each vendor had to submit its completed platform for testing. Wherever possible, the MIA conformance tests were based on existing industry tests created by organizations such as the National Institute of Standards and Technology (NIST) and the X/Open Company. After passing each basic test, for example, proving conformance to ANSI C, a vendor had to pass an additional test for the "MIA delta," i.e., for the part of the specification that was different for MIA. In general, this difference consisted of Japanese language character support and more restrictive interpretations of a specification's optional or undefined parts. In the case of STDL, however, a wholly new suite of tests was needed to confirm conformance to the basic specification.

It became clear during this stage of the project that problems existed with the way in which the solutions had been specified. For example, the specifications for new TP technology had used existing standards specifications as models. In its eagerness to accomplish the task, the MIA consortium employed traditional methods of compromise and ambiguous wording to obtain agreement among the participating vendors. Not until the conformance tests began did the problem become apparent.

The conformance tests for STDL were divided into syntax verification tests and semantic tests. Conformance testing for any language is a tremendous undertaking because there are so many potential combinations of language syntax and semantics to take into account. The first problem for NTT was to reduce the number of tests to a practical amount, while keeping the results of the tests meaningful.

Initially, NTT took the approach of translating the specification's syntax rules into syntax tests and the general rules into semantic tests. The syntax tests were designed on the assumption that a vendor's STDL compiler would produce an error message for each violation of a syntax rule. The semantic tests assumed that a vendor's run-time system would produce an error message for each violation of a general rule. The

specification had not been written using the same assumptions, however, and many of the syntax and general rules for the language elements contained a high degree of ambiguity concerning whether the rules had to be enforced at compile time or at run time.

Although this problem was never resolved for the STDL conformance tests, the tests were successful after they were redesigned to be more flexible in the method of catching errors. NTT was able to carefully monitor vendor implementations for consistency and compatibility.

MIA Applications

The intention of the MIA was to provide compliant software as the base, or heart, of a new application. MIA specifications standardize the most important interfaces and, consequently, enable users to realize the benefit of lower procurement costs, lower training costs, etc.

The MIA initiative was different from usual standards activities in that the implementations of the specifications were monitored by the same authority that caused the creation of the specifications in the first place. NTT bought systems based on its specifications, and worked with the vendors to maintain the specifications to correct problems that arose during implementation and application development.

For Digital, complying with the specifications meant implementing software to meet the terms and conditions of a large contract based on the specifications. Of course, the specifications covered only a portion of the overall platform and consequently did not address many conditions of the contract, such as CASE tools and system management.

Even though Digital's contract was for a single-vendor application, the source code had to be portable in case NTT decided to substitute another vendor's hardware for Digital's. Also, the new MIA-compliant LMS application had to fulfill at least the same functions as the old application. This application was therefore a good test of the MIA specifications; it would show how well the user requirements had actually been represented and met.

For Digital, the effort required delivering, for the first time, an integrated set of standards-compliant products for a large-scale business application. Digital had to combine components from a wide variety of internal product groups, make them all work together, and then upgrade or enhance the products to meet the MIA-specific requirements. In general, this entailed ensuring that our products were adapted to the Japanese market, i.e., that they supported the Japanese language character sets. In addition, the MIA required the integration of other new open technology, such as the RPC and other elements of OSF's DCE, DECmcc, and the new, ANSI-compliant version of DEC C.

Conclusions

Following the success of MIA, the MIA specifications became base input documents for the SPIRIT consortium, at which the user-driven standardization effort continues. Also input to SPIRIT were documents from AT&T, BT, Bellcore, and ETIS. The consortium model reduces vendor disagreements and yields a solution based on business requirements rather than on choice of vendor.

The fundamental requirement of the MIA was for a common computing platform for NTT's new enterprise applications that could be multisourced. This fundamental requirement is shared by the SPIRIT members, who represent the world's largest telecommunications corporations.

MIA and SPIRIT are seeking to lower costs in what has traditionally been the highest margin, lowest volume area of computing. The ultimate goal of a single, integrated platform that can be purchased off the shelf from a significant number of vendors does not appear to be completely attainable. Partial gains are more likely, as in the case in which suppliers integrate more or less dynamically the components of the required platform or platforms. Ultimately, the industry will be changed by the MIA and SPIRIT initiatives, although probably not in the exact way it was originally envisioned. For instance, since the MIA initiative began, the vertically integrated computer manufacturer, i.e., the manufacturer who supplies all the hardware and software components of the platform, has nearly vanished.

In the users' ideal vision, the software components conforming to the specifications in the MIA and SPIRIT platforms are off-the-shelf products that fit together easily. This goal has not proved to be the case in Digital's experience. Special product source code modifications were often required, and such modifications created integration challenges for Digital. For example, a special version of the DCE interface definition language (IDL) compiler was necessary to support the MIA. The new version mapped Kanji character set encoding to the ISO ASN.1/BER standard, whereas DCE RPC normally uses Numeric Data Representation (NDR) encoding.[26,27]

A paradox in the user-driven standardization effort derives from the fact that the MIA and SPIRIT platforms are intended for large projects, which are by definition limited in number. Therefore, creating off-the-shelf versions may be difficult due to limited platform volumes based on demand. For a vendor such as Digital, the effort appears to be best handled as a long-term partnership with large customers, supplying base technology and components to be integrated with those of other vendors. Integration becomes a continual and dynamic process. The key problem becomes systems integration, and a key question becomes who

among the multiple vendors involved in supplying components will perform the integration.

The systems integration issue, therefore, is more important than ever before. As more and more vendors, pursuing their own core competencies, develop standards-based components, the greater the problem of component integration for customers who seek large-scale application solutions becomes. Enterprise-level platforms of the future are less likely to have components that are supplied entirely by a single vendor, and large applications, even standards-based applications, will continue to require platform customizations to meet the demanding requirements of these large users.

Acknowledgments

Thanks to Roger Baust, Bob Bell, Peter Conklin, Bo Erden, George Gajnak, Bob Howell, Bob West, and Steve Young for their help in delivering the MIA and for leaving a legacy of learnings.

References

1. *Multivendor Integration Architecture, Division 1, Overview, Technical Requirements* (Tokyo, Japan: Nippon Telegraph and Telephone Corporation, NTT Data Communications Systems Corporation, IBM Japan, Ltd., Digital Equipment Corporation Japan, NEC Corporation, Hitachi, Ltd., and Fujitsu Limited, 1991).

2. *Network Management Forum Proceedings, SPIRIT Tracks, General Meeting,* Marne La Valee, France (October 1994).

3. *SPIRIT Platform Blueprint,* SPIRIT 2.0, vol. 1 (Reading, U.K.: X/Open Company Ltd., Network Management Forum, 1994).

4. P. Conklin and E. Newcomer, "The Keys to the Information Highway," *Future of Software,* Chapter 3, D. Leebaert, ed. (Cambridge, Mass.: MIT Press, 1995).

5. *Multivendor Integration Architecture, Concepts and Design Philosophy* (Tokyo, Japan: Nippon Telegraph and Telephone and NTT Data Communications Systems Corporation, 1989).

6. R. Baafi, J. Carrie, W. Drury, and O. Wiesler, "ACMSxp Open Distributed Transaction Processing," *Digital Technical Journal,* vol. 7, no. 1 (1995): 34–42.

7. *Information Systems—Programming Language—C,* ANSI/ISO 9899 (Revision and redesignation of ANSI X.3159-1989) (New York: American National Standards Institute/International Organization for Standardization, 1989).

8. *Programming Languages—COBOL,* ISO 1989:1985 (Endorsement of ANSI X3.23-1985) (Geneva: International Organization for Standardization, 1985).

9. *X/Open CAE Specification,* C192 ISBN 1-872630-09-X (Reading, U.K.: X/Open Company Ltd., 1991).

10. *X/Open Portability Guide (XPG3),* ISBN 0-13-685868-6 (superseded by X/Open C, C214, ISBN 1-872630-39-1, COBOL dropped in latest version) (Reading, U.K.: X/Open Company Ltd., 1989).

11. J. Gray and A. Reuter, *Transaction Processing Concepts and Techniques* (San Mateo, Calif.: Morgan Kaufmann, 1993).

12. *Information Technology—Open Systems Interconnection—Distributed Transaction Processing—Part 1: OSI TP Model,* ISO/IEC 10026-1:1992 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1992).

13. *Multivendor Integration Architecture,* Vol. 8, Div. 3, *Systems Interconnection Interface Specifications,* Part 4, *Remote Task Invocation Service Definition and Protocol Specification* (Tokyo, Japan: Nippon Telegraph and Telephone Corporation, 1991).

14. *X/Open Preliminary Specification, Distributed Transaction Processing: The TxRPC Specification* (Reading, U.K.: X/Open Company Ltd., 1993).

15. P. Bernstein, P. Gyllstrom, and T. Wimberg, "STDL—A Portable Language for Transaction Processing," *Proceedings of the Nineteenth International Conference on Very Large Databases,* Dublin, Ireland (1993).

16. E. Newcomer, "Pioneering Distributed Transaction Management," *Bulletin of the Technical Committee on Data Engineering,* vol. 17, no. 1 (New York: IEEE Computer Society, March 1994).

17. *Information Technology—Database Languages—SQL,* ISO/IEC 9075:1992 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1992).

18. *Information Technology—Programming Languages—FORTRAN—Extended,* ISO/IEC 1539:1991 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1991) and ANSI X3.198-1992 (New York: American National Standards Institute, 1992).

19. *OSF/Motif Style Guide,* version 1.2 (Cambridge, Mass.: Open Software Foundation, 1992).

20. *Message Handling System and Service Overview—Data Communications Networks and Message Handling Systems,* Recommendation X.400-89 (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], 1989).

21. *Information Processing Systems—Open Systems Interconnection—File Transfer, Access, and Management,* ISO 8571-1:1988 (Geneva: International Organization for Standardization, 1988).

22. *Interface between Data Terminal Equipment and Data Circuit-terminating Equipment for Terminals*

Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuits—Data Communication Networks: Services and Facilities, Interfaces, Recommendation X.25-89 (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], 1989).

23. *ISDN, I-Series Recommendations* (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], 1989).

24. *Information Technology—Local and Metropolitan Area Networks—Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications,* ISO/IEC 8802-3:1993 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1993) and IEEE 802.3-93 (New York: The Institute of Electrical and Electronics Engineers, 1993).

25. P. Conklin, "Enrollment Management, Managing the Alpha AXP Program," *Digital Technical Journal,* vol. 4, no. 4 (Special Issue 1992): 193–205.

26. *Information Technology—Open Systems Interconnection—Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1),* ISO/IEC 8825:1990 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).

27. *Information Processing—Representation of Numerical Values in Character Strings for Information Interchange,* ISO 6093:1985 (Geneva: International Organization for Standardization, 1985).

Biography

Eric A. Newcomer

Eric Newcomer is a member of the Corporate Standards Group at Digital Equipment Corporation. As Digital's primary representative to the SPIRIT consortium in the United Kingdom and former representative to the MIA consortium in Japan, he works with representatives from other computer companies to create specifications for open systems software under the sponsorship of large information technology users. Eric joined Digital in 1984. He has 17 years of experience in database and transaction processing software. He holds a B.A. in American Studies from Antioch University.

James R. Kirkley
William G. Nichols

Integrating Applications with Digital's Framework-based Environment

Digital has developed the Framework-based Environment to address the integration and interoperability needs of manufacturing and other business systems. FBE consists of a method for integrating existing applications, frameworks of industry models, and tools that use Digital's CORBA-compliant ObjectBroker integration software to manage the exchange of information between cooperating servers on the network. Using these products, Digital Consulting and its partner systems integrators provide FBE application integration services to large organizations.

The increasing quality and cost-effectiveness of computer application software has revolutionized the way organizations share and manage their information. Rather than develop custom information systems with their internal programming staffs, many businesses now purchase software available in standard "off-the-shelf" packages. A well-chosen standard package can save development time and cost. Before it can be useful, however, it must be integrated with other new software and with the mature (legacy) applications that hold current business data and processes.

Application integration can be a substantial effort. If business changes are not anticipated during the planning phase, an integrated system can be inflexible. The existing applications, both legacy and new, rarely meet current requirements. An ad hoc integration that starts with the existing applications' interfaces will seldom be flexible in ways that accommodate future business changes without widespread program changes.

An integration derived from a clear model of current and expected business processes provides a basis for growth and flexible change. Digital has developed the Framework-based Environment (FBE), consisting of reference models, methodologies, and a toolkit. Together, these products provide flexible systems integration.

In this paper, we provide a brief overview of FBE and characterize the projects that can benefit from using it. We describe flexible application integration and the benefits of model-driven integration. Finally, we discuss our experience using FBE.

Overview of the Framework-based Environment

FBE consists of the following components.

- MethodF is an object-oriented methodology based on two systems integration methodologies recognized in the industry: Jacobson's use case analysis and Rumbaugh's Object Modeling Technique.[1,2,3,4] These methodologies are explained in the section Model-driven Integration with FBE.

- ObjectPlus is a modeling tool from Protosoft, Inc. that has been tailored for MethodF with an FBE-specific code generator. In addition to the methodologies described above, the tool has extensions that provide the ability to create an implementation model. The implementation model describes how objects are distributed among the various applications.
- ObjectBroker, Digital's object-oriented integration software product, is compliant with the Common Object Request Broker Architecture (CORBA) specification from the Object Management Group (OMG).[5,6]
- A suite of supporting libraries and tools includes reference models and associated code libraries that have been abstracted from previous projects and made available for reuse. The reference models and associated code libraries are organized into frameworks of industry-oriented business objects, as given in Table 1.

  The tools include two important components: (1) The FBE Design Center is an extensible workbench architecture that supports the analysis, design, and implementation of CORBA-based distributed object systems. (2) The FBE Adapter Development System, which fits into the FBE Design Center, automatically generates CORBA- or ObjectBroker-compliant code and the necessary files to compile and link the code into platform-specific executables.

Integration Projects Appropriate for FBE

Any integration project automates previously manual processes involving existing applications. FBE and its flexible approach to systems integration allow a business to replace or add component applications efficiently as business conditions change.

FBE provides the most benefits when many different kinds of well-defined business transactions occur between a mixture of commercial and custom applications. Not all projects can benefit from FBE or its style of development. For example, if the primary task is to integrate data sources for decision support, a database integrator or a data warehouse may solve the problem quickly. If a company is not trying to gain an advantage by automating accounting more cheaply or completely than its competition, an off-the-shelf accounting package may be the right choice. At the other extreme, if the task to be automated is completely new, there may be no appropriate packages available, even as components of an integrated solution. New development would also be preferable if high-performance or real-time operation were more important than the flexibility to plug in existing, unmodified applications.

As an example of an appropriate FBE integration, consider a manufacturing operation automating a manual procedure that collects orders from an order processing system, schedules production runs, and passes the schedule to the manufacturing floor. In this example, the company wants to obtain a competitive advantage by dynamically rescheduling production based on new customer orders, at once reducing inventory costs, and improving delivery performance. This is more than a decision support system: the integration requires that applications interact with each other. Although finding a turnkey package that can operate the entire factory is unlikely, factory scheduling applications are readily available. Buying one would be more cost-effective than writing one in-house. The project would then need to integrate the legacy order processing system with the newly purchased scheduling application. The order processing system is too important to the company to risk modifying it significantly at the same time as introducing new automation.

After the integration project has been completed, though, the order processing system might be made more cost-effective by moving its function from a mainframe application developed in-house to a standard client-server product. Perhaps business conditions will have changed and the order processing system needs to be augmented so customers can submit orders directly by electronic data interchange (EDI). The project manager might decide to purchase an EDI processor to augment or replace the existing order processing system.

Later, after the manual processes have been automated on the factory floor, another project could extend the integration to send the schedule directly

Table 1
Frameworks of Industry-oriented Business Objects

Base Business Models	Manufacturing Business Models	Industry Business Models
Activity management	Order management	Semiconductor
Production management	Schedule management	Oil and gas
Resource management	Product management	Pharmaceutical
	Process management	Batch process
	Quality management	Banking and finance

to factory cell controllers. Then, if a more efficient scheduling package becomes available, it could be substituted for the older one. The modular design of FBE would minimize the programming changes required for this substitution and give the organization the flexibility to use the most cost-effective solutions.

Model-driven Integration with FBE

An integration project needs a clear process and a means to avoid being biased by the assumptions built into its component applications. We use object modeling to plan and document an integrated system in a uniform manner. The abstraction inherent in object modeling hides detail. This makes the model meaningful and allows modeler and client alike to ensure that the model matches the intended business processes. The abstraction also helps to separate the interface from the implementation. The interface describes *what* is to be done; the implementation describes *how*. The *what* of a business process changes comparatively little over time: a factory takes orders and schedules production runs, a stockbroker trades stock, a mail-order business ships packages. The *how* changes dramatically from year to year.

In the following sections, we trace the steps of a typical systems integration project as conducted by Digital Consulting or by Digital's partner systems integrators. We show how a modeler might use the FBE method, tools, and frameworks to provide application integration services.

Object Modeling

Before we start object modeling, we ensure that a business process model, or its equivalent, is completed. Sometimes a business process model results from a formal business process reengineering. More often it comes from a less formal understanding of existing processes and required changes. In both cases, the modeler will cooperate closely with someone who understands the process well. As always, the better we understand our goals, the more likely we are to achieve them.

With this knowledge, we can start FBE's object-oriented analysis and design process, known as MethodF. MethodF begins with Jacobson's use case analysis method. A use case traces a chain of events initiated by a single person (or other entity), acting in a single role, as he, she, or it works through some task. For example, we might trace what happens when a customer calls an order desk through the clerk's responses, catalog checks, inventory checks, order placement, picking list generation, and finally, package shipment. As we do this, we note all the objects and the uses that the actors make of them. Then we follow another use case. Perhaps this time the customer asks for a product that is out of stock. We follow the discussions about back-ordering and price guarantees that will make our business attractive to this customer. After analyzing many use cases, we have a list of *business analysis objects* (objects that describe requirements in business terms) and a list of the functions and attributes of each object.

We then compare the analysis objects with the *business design objects* in FBE's reference model library. Here, we may well find similar objects that use different names and detailed constructs to describe the same functions and attributes. The next step in MethodF is to merge these design objects into the model. By using objects from the reference library, we take advantage of previous modeling experience built into the reference models and prepare to reuse code associated with the reference models as well.

We use the ObjectPlus modeling tool to capture use cases in diagrams according to Jacobson's conventions. We prefer the Rumbaugh Object Modeling Technique (OMT) notation, however, for describing the business objects. OMT diagrams, with FBE extensions, define objects and the interfaces between them in enough detail that a tool can use them to generate interface definitions that can be compiled. The ObjectPlus tool also captures OMT diagrams.

A direct connection exists from the use case models, through the business models, to the design models, and to the code. We use the term *model-driven* to describe the FBE approach, because necessary changes are first made to the models and new code is then generated from the models.

Generating Interface Code

Once we have completed the design objects, we use FBE tools that work with the ObjectPlus modeling tool to generate CORBA Interface Definition Language (IDL) from the design object definitions.[6] We chose CORBA because it is an emerging industry standard designed to build distributed object-oriented systems that include existing non–object-oriented applications. A CORBA implementation, such as Digital's ObjectBroker product, generates interface stub routines that marshal data to be sent to an object, whether the object is on the same computer or across a network. For example, the stubs convert integers sent from big-endian to little-endian computers. A CORBA implementation also provides an object request broker: a run-time library that routes requests to objects in a distributed system. This allows applications running on different systems to communicate without the need for applications to know which systems will be involved.

We use the IDL interface definitions to guide programmers as they develop *adapters* between this object interface and the existing application's interface. For example, an existing program might take its

input as a formatted file and deliver its output in another type of file. Since the rest of the integration should not know about these files or their formats, we write an adapter that translates between these files and the methods and attributes of the objects defined in our model. Perhaps an alternative application uses a remote procedure call for I/O instead of the files our existing application uses. When we replace the existing application, we write new adapters using the same object interfaces. As a result, the rest of the integration needs no changes. Writing these adapters is not necessarily easy; application integration requires substantial effort, whether the integrator uses FBE or not. By restricting the changes to a single module, FBE minimizes the development and testing effort required to replace component applications.

We usually write the adapters in C, rather than C++ or a pure object-oriented language, because much of their interaction is with the applications being adapted. The existing applications were seldom built with object-oriented principles. In many cases, useful tools such as database translation programs and "screen scrapers" are available to communicate with applications that expect terminal I/O. These tools also were seldom built for object-oriented languages.

In some cases, an adapter needs to be so large that it is a small application in itself. In these cases, we might use an object-oriented language for the bulk of the code. A factory scheduler might generate production tasks based on a customer order, but the cell controllers in the factory might expect only a single task for each type of part produced. The adapter needs to combine the tasks for a given part type from several orders before it sends a message to the cell controller. As the cell controller reports progress on each task, the adapter allocates completed parts to the original customer orders. The cell controller simply makes parts, the factory scheduler simply fulfills orders, and the adapter bridges the gap between them.

Reference Models

As we gain experience working with integrators, we abstract and merge the models they build into reference models for the various application domains, such as discrete manufacturing, process manufacturing, and financial services. We collect and tailor the reference models to comply with accepted industry standards such as ISO STEP in the manufacturing domain and ISA SP88 in the process industry domain.[7,8] These reference models allow FBE modelers to build on previous experience. Even if they cannot use the reference model in its entirety, they can use it as a guide to save time and to check their own model for completeness. We also collect the adapters for frequently integrated applications into a library. Later, when we reuse a reference model, we will have corresponding adapters that can also be reused, usually after modification. It is important to note that anyone—Digital, the systems integrators (Digital's partners), and, most importantly, the customer—can build their own reference models.

From Applications to Objects: Experience Gained

Design always involves trade-offs between competing requirements. The trade-offs in an integration project are somewhat different from those in a new development project: an integration project must take existing applications into account while trying to implement a business model faithfully.

In this section, we discuss trade-offs due to the change from a functional view to an object view, then explore three familiar design topics from the point of view of an FBE integration project: top-down versus bottom-up design, improving reliability, and improving performance.

Overcoming the Legacy of Functional Decomposition

The challenge of object-oriented application integration is to make application programs, which are designed around individual business *functions*, support the unified business *object* model.

Figure 1 illustrates a sample mapping of business objects to application functions. It shows the logical objects of customer, product, and shipment, with their data structures and methods mapped to the several different application functions of transportation, warehousing, and billing. As the integration project maps business objects to application functions, it must

- Establish routings of requests for individual attributes or operations of an object to the applications that contain them
- Provide mechanisms to maintain consistency when multiple applications require the same data

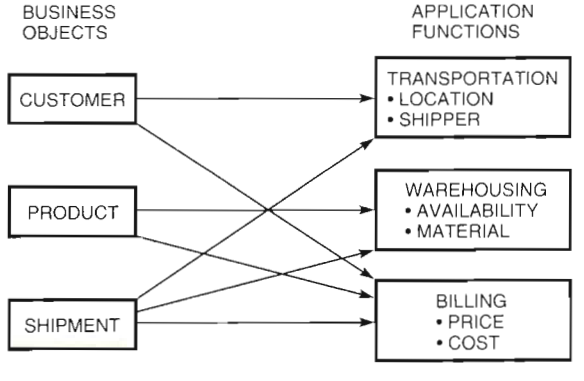


Figure 1
Sample Mapping of Business Objects to Application Functions

Split Instances When we develop the business object model, we may discover that a single logical object may be hosted (its underlying data structures and methods implemented) by more than one physical application. For example, a product object's *price* attribute is hosted by a billing application, and its *availability* attribute is hosted by a warehousing application. When we integrate these applications according to a business object model, we achieve a single logical object whose data and methods are stored in different physical applications and often in different locations. This is called a *split instance.*

When a client application requests the product's availability, the object request broker sends the request to the warehousing application and forwards a request for the price to the billing application. The requester neither knows nor cares where the information is held.

The notion of the split instance is a central principle of FBE. It allows us to model the business logically and independently of the way applications may implement business functions. The split instance is not without its problems: Many times the same information is stored in more than one application. In the above example, it is likely that both the manufacturing and the billing application maintain the product *name* attribute. Many other attributes are potentially duplicated as well. When an attribute of a type exists in two or more applications, the designer is faced with two questions:

1. When a *get attribute* operation is requested, to which application should it be delivered?
2. When a *set attribute* operation is requested, is it necessary to update only one or more than one application's data?

We cannot answer these questions in a general way, but we can highlight some points to keep in mind when addressing them.

- Get attribute. Can one application be considered the primary source for data about an object? Before any integration was in place, legacy systems provided a formal or informal process that updated secondary information sources from a primary source. The requirements statement is a good reference here. The designer should discuss this with the business domain experts to understand the way data is maintained and distributed. The primary application is the best source for such data. As a backup, secondary applications could serve as sources for the information. The designer should consider the effect of stale information on the operation of the business.
- Set attribute. When attributes are *set,* should all applications be updated simultaneously? Usually a category of infrequently changed "reference data" is accessible. The reference data is more often added to than changed. Changes to this kind of data essentially ripple through the company. Sometimes it is the slow communication of these changes throughout the organization that drives the requirements for integration (the push-pull phenomenon).

When we must guarantee simultaneous changes to data on multiple heterogeneous computing platforms or between applications that hide their data, we would prefer a two-phase commit transaction between dissimilar databases. Unfortunately, nothing is commercially available today (June 1995) that works on an arbitrary combination of databases and applications. Several products support a limited set of third-party databases and applications. If these products cannot address the need, and our applications require multiple application transactions, we may have to write the two-phase commit code.

As an alternative, we may be able to use a workflow to manage the update of several applications. An operation can be defined that is implemented as a workflow script. The workflow script can, in turn, perform the update (through additional method invocations) on the data stored in a number of different applications. This is probably closer to the customer's method and would be easily automated. A workflow capable of doing the update must have the capability of compensating for failure to update all applications. A workflow update is different from two-phase commit, because the data in the applications may be inconsistent for a brief time.

To our knowledge, Digital's ObjectBroker integration software is currently the only CORBA implementation that is able to route requests for a single object to multiple servers.

Bypassing Legacy Applications Sometimes it is tempting to bypass a legacy application and access its database directly from an adapter. The application may have a particularly difficult interface, or the required function and data may be a small part of a monolith. For simple applications, bypassing may be appropriate, but for most we must either use the application through its intended interface or replace it entirely.

The use of a legacy system to change data or perform a function can produce unwanted side effects that are not appropriate in the context of the integrated system. For example, most legacy applications maintain the referential integrity of their data through code. Invoking the database directly to add, update, or delete data risks violating this integrity.

Bypassing the application is also dangerous because changes may occur when the application is revised. Typically, application developers feel free to change the underlying data structures as long as the functionality at the user interface or formal program interface is maintained.

Top-down versus Bottom-up Design

Tension always exists between the goals of top-down and bottom-up designs. The FBE emphasizes top-down modeling; it starts with the analysis of use cases and then defines business objects independently of any existing applications. This keeps the design focused on the business problem and enhances the flexibility of our integration. We find that the most common modeling error is to accept an existing application's "myopic world view" without considering the overall system's needs. Usually, existing applications are a poor source for business object models, since many no longer represent desired business processes.

If we are not conscious of bottom-up demands on our design, however, we can design a system that requires needlessly large, complex, or slow adapters between the existing applications and our ideal model. Though we have no easy guidelines for balancing the top-down and bottom-up demands, some issues are encountered repeatedly.

The problem of partial implementations provides a simple example of this balancing requirement. Projects that use top-down modeling to derive their object models sometimes encounter a dilemma: attributes and operations appear in the model that no application in the network can implement. It is reasonable, for example, for the object model of a factory floor conveyor to define a stop operation, but the device control software installed in the factory may not provide an equivalent function.

When implementers cannot support a model, they have two choices:

1. Modify the model to reflect the capabilities of the environment.
2. Implement only the part of the model that is feasible.

The first option appears to be the easier choice, but it limits the reusability of models and diminishes the effectiveness of the top-down approach. A top-down model of the conveyor should capture the business users' expectations; implementations may or may not meet these expectations. A partial implementation simply returns an error whenever a user accesses an attribute or invokes an operation that is not supported.

The partial implementation of a conveyor can still be substituted for a complete one, though the partial one always fails when a user sends a stop request. The system must be prepared to receive an error response from an operation invocation at any time; other errors could occur during the stop operation's processing, even if the implementation were complete.

A partial implementation opens the way for subsequent versions of the software to support the feature. It provides a placeholder for an attribute or an operation and preserves the integrity of the object's specification.

Improving Reliability

Finding bugs in an integrated system is often difficult. Even if we assume that the component applications work perfectly, bugs can arise from mismatches between the components. This commonly comes about because of inconsistent business rules between applications: what is allowed in one application may be illegal in another.

An adapter in an integrated system must be a firewall; that is, it must limit the spread of errors and misunderstandings from its application. We code pre- and post-condition checks around calls to component applications. This is helpful if we code for the right conditions and leave the checks in the production code. The use case analysis and business object descriptions sometimes suggest conditions to test, but this process is informal. We find that we need more run-time checks in adapter code than in individual applications.

We also need a way to isolate a suspect application from the integrated system so we can see how the integrated system behaves without it. FBE's Adapter Development System can generate simple stubs from an object's OMG IDL. The tool generates a client stub that makes appropriate requests and a server stub that echoes its input. The stubs are simple enough to be checked at a desktop device to ensure that they work as expected. The stubs are also useful as templates for starting new adapters.

Improving Performance

Without planning and careful monitoring, a large system of dissimilar applications can be slower than the performance of the component applications would suggest. We have used standard approaches to improve and monitor performance. It is worth noting here how these approaches influence FBE design and development.

Performance Requirements in Large Systems There is often a trade-off between performance and flexibility. Our integrated system would be ideally flexible if it made separate calls through an adapter to a component application for every datum in every different circumstance. We could change storage and behavior almost with abandon. On the other hand, if each adapter were an entire rewrite of its underlying application, we could, in principle, store and manipulate each datum in the most efficient way for all accesses.

Although FBE is designed for systems that require flexibility at the cost of some performance degradation, we must be careful to deliver satisfactory performance. In the following subsections, we discuss the trade-offs in caching and object granularity.

Caching Applications frequently generate large quantities of output in response to a command, rather than the fine-grained results that are appropriate to object-oriented requests. It is often appropriate for an adapter to return only a small part of the data it receives from an application interaction and cache the rest for future requests. Applications that produce data in batches typically do not modify their state for long intervals, so the cached values remain valid long enough to be useful. Of course, there must be a means to invalidate the cache. In some cases a timer will suffice; in other cases an event, such as a new batch run, must be extended to invalidate the cache.

Adapter caches greatly improve performance and can give the adapter developer the freedom to organize and present the data in a form appropriate to the object model.

Object Granularity Designing objects that work well in a distributed system is important to ensure flexibility. Parts of a distributed system frequently move from one computer to another. We should not expect our objects or their underlying component applications to remain in one particular place.

In a pure object-oriented system, for example the Smalltalk language, everything is an object. In distributed systems, operations on objects potentially involve interaction across a network and incur network overhead. Therefore, it is not practical for everything to be an object. Some business objects will be implemented as CORBA objects (those that have object references) and other business objects will be implemented as user-defined types (passed by value). This defines the *granularity* of the object model. The decision to implement a business object as a CORBA object or as a user-defined type involves balancing flexibility with system performance.

There are no hard and fast rules that determine the most appropriate granularity for an object model. Decisions need to be based on users' interactions with the system and on the way applications use the objects they share or exchange with each other. Several matters should be taken into account when determining the model's granularity.

As an illustration, let us consider a client application that needs to display a collection of customer names in a list box. The client sends a request for these names to an object instance called CustomerList; the client and object happen to be on different computers.

In Case 1, the customer is a user-defined type represented as a C structure: it is passed by value and has no object reference. Customer attributes are stored in a CORBA-defined structure that the client code must access directly. In this case, the display of customer names may be accomplished in a single request, e.g., getCustomerNames(aCustomerList). All customer names would be passed by value. Figure 2 depicts this scenario.

In Case 2, the customer is a true object: it has an object reference and a set of attributes. The client calls the server separately for each attribute; thus the client is less dependent on the server's storage structure or any changes to that structure as it is modified in the future. In this case, a sequence of customer object references would be passed, e.g., getCustomers(aCustomerList). The client application then must request getName(aCustomer) for every customer object in the sequence. (See Figure 3.)

Clearly, the first case is more efficient in terms of network utilization; only one request is required. The second case requires $1 + n$ requests, where n is the number of customers. The first case is also more efficient at the server. Case 1 requires one database query to construct the name list, whereas Case 2 requires a separate database query for each customer.

At first glance, Case 1 would appear to be the easy winner in terms of efficiency and effective utilization of the server. This outcome, however, is not always true. Let us assume that the client application allows the user to choose from the list of customers and then displays attributes *address* and *accountStatus* for the selected customer. Here, we are faced with a choice between performance and flexibility:

1. The client could make another request that would return *all* information about a customer in a structure. Then the client application could sort through this information and display the required data. The performance is good: one request and database query provided all the data the client could want. Unless the volume of data is very large, sending the data in one message yields better

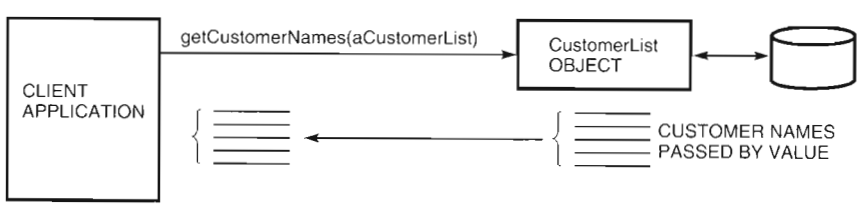


Figure 2
Case 1: User-defined Type

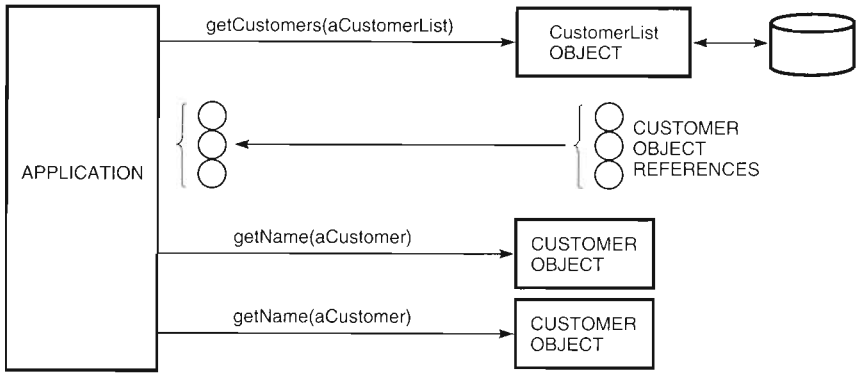


Figure 3
Case 2: True Object

performance than sending multiple messages for a subset of the data. On the other hand, this approach is inflexible: if the server changes the structure it uses to represent this data, all client software that reads the structure must change as well.

2. The client could make separate requests for each field. If the server returns an opaque object reference along with each customer's name, then the client can send a request asking for the specific fields it needs. The performance is worse than in Case 1, of course, because of the extra network traffic and message parsing. However, this approach is flexible. Since the client never looks in the object reference (it is opaque), we preserve the server's flexibility to use any data needed to retrieve the appropriate record. As long as the server continues to support the fields the client requires, the server finds them in its own database no matter how the storage structures have changed.

To ensure that the system provides the maximum flexibility, the designer should consider the following guidelines.

- Start with a fine-grained approach for modeling.
- Implement the approach using fine-grained methods.
- Change to a coarser grain if performance is an issue.

Summary and Future Directions

Developing integrated applications is not always a straightforward process. The applications being integrated are seldom an exact fit to their assigned roles in an integrated system. If they were, we would probably be able to purchase the integration from one or more of the vendors who had engineered the fit.

Integrated systems built with FBE are clearly documented with Jacobson use case diagrams, Rumbaugh OMT object diagrams, and OMG IDL. The existing applications are used indirectly through object interfaces and adapters, so the rest of the system can address them as if they were the ideal business objects modeled in the OMT diagrams. We call them business objects to emphasize their distinction from objects defined or implied by the existing applications.

The adapters are constrained by the interfaces that FBE generates automatically from the business object representations, so they do not stray from the models that document their behavior. Adapters are not always easy to write; they can be quite difficult, depending on the existing application's fit with its intended use. By restricting this awkward code to object adapters, we keep the overall integration modular. Thus we give an organization the flexibility to use the most cost-effective systems as business conditions change. We build on our experience by collecting reference models that help us to reuse the best models and adapters.

FBE continues to evolve rapidly, with improvements in the reference models, the tools, and the support for adapter writers. For example, developers have asked for better integration between the Jacobson and Rumbaugh models, between the modeling tools and the code generation tools, and for reliable queuing and workflow as well as CORBA communication between objects. In response to these requests, we now provide better integration between the analysis, design, and implementation portions of the FBE life cycle as well as code generation for trace messages and support for management and debugging of the runtime system. We would like to organize the reference libraries into pairs of object models and corresponding modules (applications and adapters) that can be assembled to build integrated applications, thus creating truly reusable business components.

We will be pursuing these and other improvements as our experience grows with integrated, distributed applications.

References

1. *The Framework Based Environment: MethodF, Version 3.0, FBE Engineering* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-QC50A-TH, 1994).

2. I. Jacobson et al., *Object-Oriented Software Engineering: A Use Case Driven Approach, 4th ed.* (Wokingham, England: Addison-Wesley Inc., 1992).

3. I. Jacobson et al., *The Object Advantage, Business Process Reengineering with Object Technology, 1st ed.* (Wokingham, England: Addison-Wesley Inc., 1995).

4. J. Rumbaugh et al., *Object-Oriented Modeling and Design* (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1991).

5. *ObjectBroker: Overview and Glossary, Version 1.0* (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-Q9KJA-TK, 1994).

6. *The Common Object Request Broker: Architecture and Specification, Revision 1.2* (Framingham, Mass.: Object Management Group, Order No. 93.12.43, 1993).

7. *Industrial Automation Systems and Integration—Product Data Representation and Exchange—Part 44: Integrated Resources: Product Structure Configuration, ISO 10303-44, WG3 N127* (Geneva: International Organization for Standardization, 1992).

8. *Batch Control Part 1: Models and Terminology, Draft 12: 1994* (Research Triangle Park, N.C.: Instrument Society for Measurement and Control, Order No. ISA-dS88.01, 1994).

Biographies

James R. Kirkley III
Jim Kirkley has been with Digital for 16 years. For the last six years, he has been involved in the development of object-oriented architectures for business application integration. A software consulting engineer, Jim is the technical director for the Applied Objects Group, which is currently focused on the development of tools and methodologies for the integration of business systems. He is the principal author of the methodology used by Digital Consulting to deliver consulting and practice systems integration using CORBA-compliant middleware. He received a B.S. in electrical engineering from Colorado State University in 1971 and an M.S. in computer science from Colorado University in 1974.

William G. Nichols
As a consultant engineer with Digital, Wick Nichols was part of a team that reviewed the Framework-based Environment and provided a report suggesting several improvements. His familiarity with related networking products, particularly DCE, enabled Wick to participate in the delivery of several FBE projects to customers. During his 15 years with Digital, Wick contributed to several projects, including the development of distributed file services. He also served as project leader of a group that developed the DECnet-10 system and as project leader and supervisor for the DECnet-20 product. He received an A.B. from Harvard University in 1973.

Owen H. Tallman

Project Gabriel: Automated Software Deployment in a Large Commercial Network

Digital entered into an agreement with a major French bank to develop an automated software deployment facility, i.e., to provide centralized control of software installations and upgrades for a large network of computer systems. Independently, Digital had developed a set of models designed to guide the design of solutions to this type of complex management problem. The bank project team, which had considerable experience building distributed system management applications, was able to take advantage of these models. The result was a versatile, scalable application for distributed software deployment, validation of the models, and a clearer sense of the usefulness of such models to complex application problems.

A large French bank purchased a DECnet network from Digital and was in the process of deploying the network to support all its banking operations. The network topology included approximately 3,000 OpenVMS VAX systems and about 18,000 MS-DOS PC workstations. As illustrated in Figure 1, these systems were arranged in a branch structure that roughly followed the geographical distribution of the bank branch offices and their roles in the branch hierarchy. At the bank's headquarters, an OpenVMS cluster and an Ethernet local area network (LAN) linked the mainframe data center with the rest of the banking network. The cluster was connected to the first tier of approximately 200 branch group servers. The second tier consisted of approximately 1,800 branches, each with between one and four branch servers, for a total of about 3,000 branch servers. Each branch server, in turn, provided Digital's PATHWORKS and application services to the PC workstations.

For its nationwide backbone network, the customer was using a public X.25 network, which was its only available option.[1,2] The cost for X.25 service was based on usage, so each packet of data transmitted increased the operation cost. Therefore, the need to minimize this X.25 expense was a fundamental factor in specifying requirements for virtually all software deployed in the network.

The bank's business depended on the correct, reliable, and efficient operation of the network. Consequently, network management was crucial. From the customer's viewpoint, such an undertaking meant management of systems and applications, as well as the communications infrastructure. By extrapolating its overall experience with the hardware deployment, and its initial experience with software deployment, the customer foresaw potentially unacceptable labor costs for software deployment using the available methods. The customer therefore gave high priority to improving the software deployment process.

In this paper, the term deployment (or deployment operation) represents a process that deploys a set of software components to a set of systems. A deployment is described by a deployment plan and requires

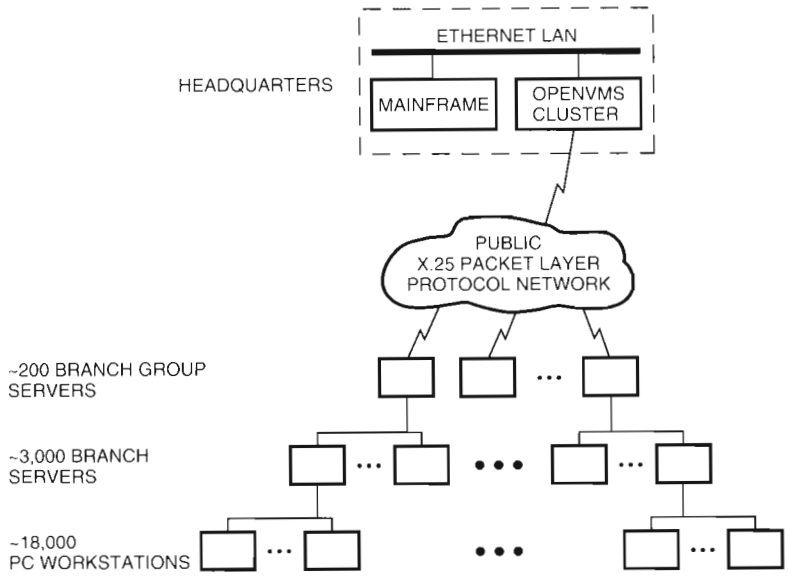


Figure 1
DECnet Network Topology in a Banking Environment

a deployment program, deployment automation software to execute the program, and an operations staff to schedule and monitor deployment program execution and, when necessary, respond to run-time problems.

The Software Deployment Problem

Ideally, the bank wanted networkwide consistency in its software, with automated, nondisruptive upgrades administered from a central point. Given the scale of the network and the number and variety of software components in use, however, this was not a realistic goal. The challenge of building a system of automated deployment tools that is capable of maintaining consistency across 3,000 widely distributed, frequently updated systems is significant in itself. Adding the problems of maintaining consistency in detailed business practices and user training in every branch greatly increases the difficulty. Actually, the business required software configurations tailored to and maintained consistently within individual business units such as branches and branch groups. Software upgrade planning and deployment activities would be essentially continuous, with numerous planning and deployment operations under way concurrently. The bank's business would not tolerate network malfunctions caused by ongoing upgrade operations or version mismatches among systems in a business unit, nor would it provide for on-site support at branches or branch groups. To implement a fully automated software deployment process would require rigorously managed, centralized planning and operational control.

The bank had already implemented a system that automated significant parts of the deployment process, using a variety of existing tools and ad hoc integration. These tools included Digital Command Language (DCL) command procedures, the Information Distribution Controller (IDC) product, which distributes files in batch mode, and a system event reporter. The process, however, was still labor intensive. The customer concluded that the only way to achieve acceptable operational costs was to increase substantially the degree and quality of automation in the process.

Customer Requirements

A solution to this software deployment problem would have to support (1) sophisticated, carefully managed planning, (2) a means of determining the current state of target systems for use in planning, (3) rigorous software certification, and (4) a highly reliable means of automating software distribution and installation. The bank's planning and certification processes were already developed, staffed, and in operation. An inventory control database for tracking system configurations was under development. However, the means to distribute and install software effectively was lacking and would have to be developed and then integrated with the other system components. The customer emphasized this need for distribution and installation automation when it first presented the problem to Digital.

All new software must be evaluated, acquired, packaged in kits that can be installed automatically, tested, and certified. Since software interdependencies may exist, multiple software components may need to be processed together to ensure proper installation and operation as a set. (In this paper, the term component refers to any software that might be distributed as a kit, e.g., a commercial layered product, an in-house application, or a patch.) Planners must determine which of the certified components to install, the branch group to install them in, and the scheduling constraints. The result is a carefully documented, uniquely named deployment plan. Deployment execution consists of performing all the steps necessary to distribute and install the software on the target group and to report the results for incorporation in the planning for the next deployment.

The operations staff, i.e., those who monitor and control the network on a continuous basis, keep a repository of data that reflects the current state of software on the systems in the network. Planners use this data to plan new states for parts of the network; they store these plans in the repository also. As many as 10 planners may be developing plans simultaneously. For each plan, an application analyzes the differences between the planned state and the current state of the network and produces a deployment program.

A deployment operation may involve multiple products. This set of products must include all those necessary to satisfy the prerequisites of the other members of the set (if they are not already satisfied by products on the target system). The members of the set must be installed in the proper order. The planners determine the proper membership for any product set and create representations of those sets in the repository. They also represent the product installation order in the repository in the form of installation precedence relationships. The deployment software uses this precedence information to determine the order of installation for members of a product set.

The operations or configuration staff store the certified software kits in a library at the management center. When the kits need to be installed on a system, the deployment software compresses the kits and then copies them across the X.25 backbone to staging areas on servers. From these areas, the deployment software copies the kits to the target system or systems or, if necessary, to servers closer to the target systems and then to the target systems, where the kits are decompressed and used. By staging kit distribution in this way, each kit is copied only once over each link, which avoids wasting bandwidth. When all the target nodes have the required kits, the kits at the staging points are deleted. The copy operations must proceed concurrently whenever possible. Table 1 shows possible states and transitions for a software component kit on a target system.

Table 1
States and Transitions for a Software Component Kit on a Target System

Initial State	Action	New State
(Null)	Copy	Distributed
Distributed	Delete	(Null)

Installation is a multistep process designed to allow the synchronized change of operating software on all related systems. Once the required kit is present on the target system, the product can be installed, i.e., the files put in place and any other necessary steps taken so that the product is ready to be activated. Activation, i.e., making the new product the current operating version, is the last step. A product can also be deactivated and deinstalled. To upgrade a product requires installing the new version, deactivating the old version, and then activating the new version. If the activation is successful, the previous version can be deinstalled. Only one version of a product can be active at any given time. Table 2 shows the states and transitions for a software component on the target system.

Table 2
States and Transitions for a Software Component on a Target System

Initial State	Action	New State
(Null)	Install	Installed
Installed	Activate	Active
Active	Deactivate	Installed
Installed	Deinstall	(Null)

Table 3 shows the state transitions to be managed between the new version product kit, the new version product, and the previous version product on the target system. Note that the deployment process should minimize the time a target system must spend in step 4, when both versions of the product are installed but neither is active.

Table 3
State Transitions to Be Managed on a Target System

Step	New Version Product Kit	Old Version Product	New Version Product
1	(Null)	Active	(Null)
2	Distributed	Active	(Null)
3	Distributed	Active	Installed
4	Distributed	Installed	Installed
5	Distributed	Installed	Active
6	Distributed	(Null)	Active
7	(Null)	(Null)	Active

A planner can specify to the deployment software that an upgrade must be carried out as an atomic transaction. That is, the activation transition must either succeed or be rolled back. In a rollback, steps 3, 4, and 5 in Table 3 are reversed. Most commercial software is not packaged with installation procedures that support installation, activation, deactivation, and deinstallation steps. Therefore, the bank must package its own software and repackage software from manufacturers so that upgrades behave this way. The deployment software invokes the individual steps by executing DCL command procedures provided in each such customized kit.

The activation of all products in a deployment may be transactional, in which case all the products must activate successfully or all activations will be rolled back. The installation steps for all the products are completed first, so all the products are ready for activation at the same time. The activations are then attempted. If all succeed, the newly activated products remain as the current operating versions. If a product activation fails, it and all the preceding activations are rolled back, in reverse order of activation, and the previous versions are likewise reactivated. When the rollback completes, the deployment stops and the management center receives a status report. Once the operations staff has corrected the problem that caused the failure of the activation phase, a new deployment program may be generated. It will execute only the activation steps, not any of the preceding steps that had succeeded. That is, the new deployment program picks up where the earlier one left off.

This transactional behavior applies to all activations across all systems in a given deployment and may involve different sets of products for different systems. The transactional characteristic applies to the deployment operation, not to a product or set of products. Thus, the deployment can accommodate interdependencies among products on different systems. If an activation of any product fails in a transactional deployment, all current or completed activations will be rolled back in reverse order of activation, regardless of location. This requirement is specifically for client-server applications whose client and server components must be upgraded both simultaneously and atomically.

The deployment software must maintain the state of the deployment in stable storage so that the state can be restored and the processing continued despite transient failures of systems or networks. The software must report the state of processing to the management center at some reasonable interval and also when the deployment completes. The software then updates the repository with the status of all the individual operations in the deployment.

The deployment implementation must provide management directives to start, suspend, resume, stop, and abort the deployment, without leaving it in an inconsistent state or disrupting business operations. Suspension prohibits any new command procedure executions from starting but does not interrupt ongoing ones, thus allowing the deployment to quiesce. Suspension does not affect transactions. The resume directive restarts execution of a deployment that has been suspended. Stopping is the same as suspension except that once stopped, the deployment cannot be restarted. The abort directive stops ongoing command procedure executions by terminating their processes and thus forces the rollback of any transaction that is executing at the time the directive arrives. An aborted deployment cannot be restarted. There is also an update directive, which forces the current details of operation state to be rolled up to the management center. A show directive reports the overall state of each deployment at a particular host.

The management directives allow an external entity, e.g., a batch scheduler or an operator, to intervene in what would otherwise be a self-contained, automated operation. A batch scheduler can suspend all ongoing deployments at some time before bank branches open and resume the deployments when the branches close. It can force a deployment to stop at a predetermined time, whether or not it has completed. An operator can use the update directive to roll up the state to determine how far a remote part of a large deployment has progressed. It can also issue suspend and resume directives to subsets of the network affected by a deployment to allow for emergency manual intervention without suspending the entire deployment.

Digital's Response to the Requirements

Digital's decision to undertake the project of developing an automated software deployment facility for the bank was based on two goals. First, Digital wanted to meet the needs of an existing customer. Second, in solving the customer's problem, Digital could validate the set of network and system management models it had already developed. The following sections provide an overview of the models and details of the automated software deployment implementation.

The EMA Configuration Management Model

When Digital began discussions with the bank about automating software upgrades, in the Enterprise Management Architecture (EMA) group, Paul Kelsey was developing a comprehensive general model of configuration management for information systems. Like the influential EMA entity model that preceded it, the EMA configuration management model (CMM)

defines a consistent set of concepts and terms for working in its particular problem domain.[3] The entity model broke new ground by applying what would come to be known as object-oriented concepts to the problem of managing the many types of objects found in a network. The CMM goes on to address the relationships among those objects that, in combination with the objects themselves, constitute an information system's configuration.

Configuration management concerns a broad range of activities over the lifetime of an engineered system. The larger or more complex the system to be managed, the greater the need for a configuration management discipline. The U.S. Air Force defines configuration management as "a discipline applying technical and administrative direction and surveillance to (a) identify and document the functional and physical characteristics of a configuration item, (b) control changes to those characteristics, and (c) record and report change processing and implementation status. It includes configuration identification, control, status accounting, and audits. Configuration management is thus the means through which the integrity and continuity of the design, engineering, and cost trade-off decisions made between technical performance, producibility, operability, and supportability are recorded, communicated, and controlled by program and functional managers."[4]

The CMM provides a conceptual framework for automating information system management, covering the entire scope defined in the preceding paragraph. For example, consider a disk drive. The EMA entity model provides a conceptual framework for describing the drive as an object with certain attributes (e.g., storage capacity) and operations (e.g., format) such that developers can build software that allows monitoring and control of the object by means of a management protocol. Any object in the network that presents a conforming management interface is called a managed object.

The CMM proposes a framework for describing the disk drive's role in a system configuration over the drive's lifetime. The framework covers

1. The services that the disk drive provides and the clients of these services, e.g., the logical storage volume that the drive supports
2. The services that the disk drive consumes
3. The objects that compose the drive
4. The drive's current and previous attribute values
5. The attribute values that the drive should presently have
6. Plans for future drive configurations
7. The way software should interpret and act on list items 1 through 6

The following discussion emphasizes the aspects of the CMM that influenced the design of the Project Gabriel software.

Persistent Configuration Model

In the CMM, all users and management applications deal with managed objects in an information system, whether physical or abstract, in the abstract: they manipulate their representations in a repository, and automatic mechanisms carry out the implied operations transparently. The repository maintains a persistent representation, i.e., model, of the entire information system's state; it is called the persistent configuration model (PCM). The PCM provides a common level of abstraction for all users and management applications because all management actions are taken through it. Since the model persists, the PCM can provide this abstraction in multiple temporal divisions.

Temporal Divisions

Managed objects indicate their state through attributes and through relationships with other objects. Object state is relative to the temporal division of the PCM through which the state is viewed. Each temporal division can provide a consistent view of all the objects in the network as they were at some point in the past, as they are now, or as they will be.

The historical temporal division records past system states. The present is represented in the observed and expected temporal divisions, where the observed division provides the most recent information available on actual object state, i.e., what is now. The observed division is populated by automated census services that collect current state information as directly as possible from the objects. The expected division maintains what is currently intended for the object state, i.e., what should be. This division is based on the observed division but modified as necessary to represent the state sanctioned by the system or network administrator.

The planned and committed temporal divisions represent future object states. States that may be realized at some time are planned, whereas those that will be realized are committed. The distinction permits simulating, analyzing, and evaluating future states in the planned division without implying any commitment to realize them.

Realization

Differences between object states in the expected and the committed divisions indicate changes that need to take place to realize the new committed configuration. This is the task of the realization services. The job of identifying the required changes and generating a program to realize these changes is called configuration

generation (CGN). Other realization services execute the program and update the repository based on the results. A software deployment operation would be called a realization in CMM terms. The ultimate vision of the CMM is to allow the user to define the desired state of an information system and, with a single command, to realize it.

Once the planned state has been realized, automated services can maintain that state by monitoring the differences between object states in the observed and the expected divisions. These differences represent possible faults and trigger fault-handling actions.

Implementation

Digital and the bank agreed that Digital would implement the critical deployment automation part of the bank's requirements and integrate it with the bank's established processes. The focus of the discussion in this section is the engineering team's efforts to arrive at an effective, implementable system design.

System Design

The CMM provided an effective conceptual framework for thinking and talking about the system requirements and possible design choices. As one would expect from a general model, however, the CMM did not address important design and implementation issues. In particular, it did not prescribe in any detail the PCM design or how the realization services should work. The Project Gabriel engineering team, which included the CMM author, had to quickly answer the following basic questions:

- How should the team implement the PCM? Is it an object-oriented database, or will it require functionality beyond what the team can implement in such a database? What schema should the team use? How much of the PCM as described in the CMM is really necessary for this project?
- How will CGN convert the PCM state data to a deployment program? Is CGN a rule-based application or a conventional, sequential program? What will CGN require of the objects in the PCM? How will CGN communicate to the other, as-yet-undesigned realization services what needs to be done to carry out a deployment? How should the team trade off the complexity of CGN versus the complexity of the services that will execute the programs?
- What services will the team need to carry out the programs CGN generates? What form will these services take?
- How can the team minimize the complexity of the system to arrive at a design that the team can actually implement?

The last question was in many ways the most important. The team had to break down the problem into manageable pieces and at the same time devise an integrated whole. The team did not have time for a sequential process of analysis, design, and implementation and, therefore, had to find ways to start development before the design was complete. CGN presented the pivotal problem; it might ultimately be the most difficult part of the system to design, but the components on which it depended had not yet been designed. In addition, these components could not be designed effectively without some reasonable idea of how CGN would work. To efficiently use the time allotted, the team began to search for the key design abstractions while it evaluated technologies and tools.

Actions and States PCM configuration data represent multiple actual or possible states of the systems in the network. CGN would generate a deployment program based on the differences between the expected and planned states represented in the repository. This idea led to the development of a state table, which prescribed the state transitions that would have to occur to change each product on each system from its present state (as shown in the expected temporal division) to its planned future state. CGN could associate an action with each transition and program those actions. When the PCM received status from the actions taken on the target systems, the transition identifier would be included and would be used to update the PCM. This became one of the key design concepts of Project Gabriel: to model the target of a deployment operation as a collection of finite state machines.

CGN needed a way to program the actions so the other realization services could carry them out. The team chose to model the actions in a consistent manner for all foreseeable variations, regardless of how they are implemented or what state change they effect, as follows:

1. All actions consist of invoking a command, with some list of arguments, on some object, and within a discrete process.
2. Actions are associated with state transitions. Actions themselves have state (e.g., running) and finite duration. Actions can be started, and at some point they complete. When they complete successfully, they change the state of an object; when they fail, they do not.
3. The implementation of the command should behave such that an action's failure has no undesirable side effects, e.g., disabling a system component or causing large amounts of disk space to be occupied needlessly. This behavior cannot actually be guaranteed, however, so some failures may require human intervention to correct side effects.

In most respects, this model of command procedure execution is the same one used by both the OpenVMS batch facility and the POLYCENTER Scheduler. The principal difference is that in Project Gabriel, a user does not simply program an arbitrary sequence of actions. Rather, each action corresponds to a specific meaningful state transition of an object. When the PCM receives completion status for an action, the PCM update program can use the transition identifier to determine what state an object has attained and modify its representation in the repository accordingly.

By hiding the implementation internals behind a consistent interface in this manner, the software designed for controlling actions does not have to be concerned with those internals. This is a straightforward application of the principle of encapsulation, which separates the external aspects of an object from its internal implementation details.[5] Encapsulation allows a system designer to separate the question of how an action, such as copying a file or invoking an installation procedure, is implemented from the question of what interface the control system will use to invoke the action. This is obviously a simplification of the implementation issue, because the team had to deal with preexisting implementations, which cannot always be made to follow new rules. From a design point of view, however, the simplification is essential.

Control Distribution A deployment operation consists of multiple actions, performed in various complex sequences. The team understood intuitively that every host system would have to run software to execute the deployment program and that the management center would distribute the program to the other host systems in the network. An advanced development team working on a more scalable design for the POLYCENTER Software Distribution product had previously developed a model for this kind of distributed control. The Project Gabriel team adopted two related design ideas from its work.

The first idea is recursive program decomposition and delegation. Assume that the control system is implemented by servers called control points, whose task it is to coordinate operations. Assume also that each target system has an agent that carries out the action. Assign to each target agent a control point, and assign to each control point its own control point, such that these control relationships form a tree structure.

Assume that deployment programs are composed of nested subprograms, which, in turn, are composed of nested subprograms, and so on. Assume also that each program (or subprogram) has an attribute identifying the designated control point to which the program must be sent for processing. Such programs can be decomposed, distributed, and executed using a recursive distribution algorithm, as follows.

An operator submits a complete deployment program to its designated control point. (Submission consists of copying the program file to a well-known place on the management center host system and issuing a RUN command with the file name as an argument.) The control point breaks down the program into its component subprograms and submits the individual subprograms to their own designated control points, thereby delegating responsibility for the subprograms. The delegation ends when a subprogram has been broken down to the level of individual actions, which are delivered to the agent on the target system for execution. In the original model developed for POLYCENTER Software Distribution, program structure did not influence how operations were decomposed and delegated. Instead, a target could be a group of targets, allowing recursive delegation of subprograms according to the nesting of the groups. The Project Gabriel innovation was to use nested subprograms within the deployment program rather than nested target groups. Both approaches are built on the notion of distributing control by following a tree whose nodes are managed objects and whose edges are control relationships. This is how they were ultimately represented in the PCM.

The second idea relates to program state. The team modeled the deployment program and each of its component subprograms as finite state machines. Each subprogram goes through a definite series of transitions from ready to completed, stopped, or aborted. The state of the program as a whole reflects the state of the processing of its component subprograms, and the state of each component reflects the state of the processing of its components, and so on. At any time, an operator can issue a show directive for a control point and determine the local state of all deployment programs. Understanding the collective, distributed state of a deployment may be difficult at times, because a given control point may have outdated information about a delegated subprogram. For example, a program may be running when none of its components are running yet, when some are running, and when all have completed but notice has not yet rolled up to the root of the control tree. This latency is natural and avoidable in such a system.

The deployment software maintains program state on disk. When a component subprogram is delegated, the state is reflected at the sender by a placeholder subprogram that stands in for the one created at the receiver. The state is updated at the sender only after the receiver acknowledges receiving the subprogram and securing it in stable storage. Given this conservative approach to recording state changes, and logic that makes redundant delegations harmless, a control point server can be stopped or restarted without losing program state.

Data Distribution The team borrowed the notion of a distribution map from the IDC product mentioned in the section The Software Deployment Problem. The Project Gabriel concept is a distribution tree, which is formed in the same fashion as the control tree. Each host system is assigned a distribution point from which it gets its copies of software kits to be installed. A system that hosts a distribution point has its own assigned distribution point, and so on, for as many levels as necessary. This assignment takes the form of relationships between system objects in the PCM. CGN uses the distribution tree to determine the software distribution path for each target system.

The control and distribution trees need not be the same, and they should not be confused with one another. The control tree uniquely defines the path by which all other services, e.g., kit distribution, are managed.

SYREAL Programming Language To communicate a deployment plan to the servers that were to execute it, the team invented a simple textual representation called the system realization language (SYREAL). This language was easy for the developers and users to analyze in case problems developed and could easily be produced by programs, by DCL command procedures, or by hand. Although SYREAL is verbose (e.g., installing a few products on a dozen systems requires hundreds of lines of text), it clearly reflects the structure of the deployment operation.

PCM Implementation The development team believed that an object-oriented repository would provide the most natural mapping of the PCM abstractions onto a data model. The team used an internal tool kit called AESM, which was layered on the CDD/Repository software product. The user interface is based on DECwindows Motif software, using facilities provided by AESM.

AESM uses membership, i.e., containment, relationships to connect objects in a meaningful way. All relationships are derived by inheritance from this basic type. Thus, the PCM contains temporal divisions, which contain groups of systems, which contain software configurations, which contain specific software components with certain state attributes. A software catalog contains configurations, software components, and materials objects that describe the kits used to install these components. A plan in the PCM is an object within the planned domain that contains systems and configurations.

Configuration Generation Processing Thus far, the paper has described the following abstractions available for CGN:

- The PCM, which contains systems and a catalog of software configurations, software components, materials, and precedence relationships—all in temporal divisions.
- Software component state table.
- Actions, which change the state of objects in the network.
- Managed objects (e.g., software components and kits) as finite state machines whose transitions result from actions.
- A control tree to partition control responsibility. This tree consists of relationships between control points and between control points and target agents.
- A distribution tree to define the path for distributing software to target systems. This tree consists of relationships between distribution points and target agents.
- Deployment programs as finite state machines whose nested structure is decomposed and distributed according to the control tree.
- Control point servers that execute deployment programs and target servers that execute actions.

Given these abstractions, the key problem of designing CGN was to determine the optimal order of traversing and analyzing an interrelated set of trees connected with a plan in the PCM. The solution had to address

- The PCM temporal divisions, to locate expected and committed states of system configurations in the plan
- The software catalog, to determine materials and precedence relationships
- The precedence relationships, to determine the processing order for the products in the plan
- The control tree, to determine how control must be distributed
- The distribution tree, to determine how software kits must be distributed

For each system, CGN must determine what products will undergo which state transitions based on the state table. The same set of abstractions made it clear what form SYREAL should take and the nature of the processing that the control point and target servers would perform.

Reducing the problem to a small number of abstractions, many of which shared a similar structure, was a major step in the process of defining an implementable system. Although the overall problem was still complex and required a nontrivial effort to solve, at least the problem was bounded and could be solved using conventional programming techniques.

Overview and Example of Deployment Processing

A user, i.e., planner, begins the deployment process by populating the repository with objects to be managed using an application that reads from the inventory database. The objects in the repository represent a software catalog, expected and planned temporal divisions, computer systems, software products, software configurations, software materials (kits), and product pick lists. By specifying the relationships between the objects, i.e., by actually drawing the relationships, the user develops a model of the network configuration. For example, a model may represent a system that has a particular software configuration and is contained in one of the temporal divisions.

In addition to allowing the user to model the network, the deployment software represents policy information by means of relationships. A software product may have precedence relationships with other software products that prescribe the installation order. Each system has a relationship that indicates its distribution point, i.e., the file service that provides staging for software distribution to that system. Each system also has a relationship that indicates its control point, i.e., the management entity that controls deployment operations for that system.

Using the graphical user interface, a planner derives new configurations from approved configurations in the repository and assigns the new configurations to systems or groups of systems. A planner can view the differences between the current and the proposed configurations and see which systems will be affected. If the observed changes are acceptable, the planner can run CGN to produce a program to realize the changes. Once the program has been generated, the planner can launch it immediately, schedule it for execution later, or just review it.

Deployment programs normally run under the control of a batch scheduler. For large-scale deployments, which can continue for days, the scheduler automatically suspends execution while branch offices are open for business, resumes execution when the branches close, and repeats the cycle until the operation has completed. Operators oversee the execution of the deployment, intervening to suspend, resume, stop, or abort the process, or to observe the program's state. Actions on individual systems that fail may suspend themselves, thus allowing an operator to intervene and correct the problem and then, if desirable, restart the operation.

Certain events, such as a deployment action failure, roll up to the central control point and trigger the execution of a user-written event script. Depending on the type of event, the script may notify an operator, make a log entry, or perform a PCM update. Normally, the last event that occurs is the completion of the program. If the PCM completed successfully, it is automatically updated. Even if a program does not run to successful completion, the operator can trigger a PCM update so that whatever changes were realized will be reflected in the PCM. A new program, generated with the same planned configuration, will include only the changes that were not completed in the previous attempt.

The remainder of this section describes the role of each major Project Gabriel component in the deployment process. The example presented was intentionally kept simple. Its assumptions are as follows:

- The repository has been populated with network information, the product catalog, etc.
- The goal is to upgrade the software configurations of a set of four branch servers, B1 through B4.
- Central control points exist at headquarters, HQ, and on two group servers, G1 and G2 (see Table 4).
- Branch servers B1 and B2 have their control point on G1; B3 and B4 have theirs on G2. HQ hosts the control points for itself and for G1 and G2.
- The branch server systems have distribution points (file servers), which in this example are on the same host systems as their respective control points. (This overlap is not required.)
- In the PCM's expected temporal division, the four systems B1, B2, B3, and B4 are governed by the same software configuration. The only layered software product is Product X version 1.1, which is in the active state.
- The planners want to have Product Y version 2.0 installed on the four systems and in the active state. They create a plan in which a new configuration, with Product Y added, governs the systems (see Table 5). They commit the plan, which invokes CGN.

Configuration Generation CGN transforms the desired future state represented in the PCM to a program that can be used to realize that state. CGN determines the difference between the configurations in the

Table 4
Designated Management Control and Distribution Points

System	Control Point	Distribution Point
HQ	HQ	HQ
G1	HQ	HQ
G2	HQ	HQ
B1	G1	G1
B2	G1	G1
B3	G2	G2
B4	G2	G2

Table 5
Expected and Committed Configurations

Temporal Division	Configuration Name	Product	Version	State
Expected	GoodConfig	Product X	1.1	Active
Committed	BetterConfig	Product X	1.1	Active
		Product Y	2.0	Active

expected and committed temporal divisions, which in the example is the addition of Product Y version 2.0 in the active state. Since the configurations differ by only one product, the question of installation order does not arise. If multiple products were involved, CGN would analyze their dependencies and arrange them in the correct installation order.

CGN uses a state table to determine the sequence of transitions that must occur to bring the software to the desired state. In the example, Product Y version 2.0 is not present on any of the target systems, so the kit must be copied to the appropriate distribution point and then copied to the target systems, after which it must be installed and activated. CGN uses the distribution tree to find the appropriate distribution points and then uses the control tree to determine which control point to use for each set of systems, for each staging copy, and for each transition. Finally, CGN generates the corresponding text in SYREAL. The program that CGN writes optimizes throughput by performing concurrent processing whenever possible.

SYREAL Program A SYREAL program has two parts: (1) object declaration and (2) the executable. The first part declares the objects to be acted upon. The control point that executes the program has no knowledge of the software products, files, kits, copy commands, etc. It knows only that objects exist that have identifiers and that undergo named state transitions as a consequence of executing commands. SYREAL provides a means of declaring objects, their identifiers, the associated transitions, and the commands that effect the transitions. Figure 2 is an example of an object declaration. The program declares the realization object that represents Product Y version 2.0. The object name is PY. Note that PY is an ad hoc, purely local naming scheme. Since there can be only one instance of any product version on a system, the name is implicitly distinguished by its locality, in the sense that it is the unique instance of product PY on system X. PY inherits the default object characteristics (not shown) and adds its own kit identifier, product name, and a definition of the ACTIVATE transition. This transition has command CMD, which is a DCL command string.

The second part of a SYREAL program is the executable. (Figure 3 shows the executable part for the deployment process example.) This part consists of at least one executable block (i.e., subprogram), which may contain any number of additional executable blocks. A block may be defined as concurrent or serial. Blocks nested within a serial block are executed in order of appearance. Blocks nested within a concurrent block are executed concurrently.

Any block may have an associated fault action expressed as one of the following commands: ON ERROR SUSPEND, ON ERROR CONTINUE, or ON ERROR ROLLBACK. A block is executed by "USING" a designated control point to control it. For example, the first executable line in Figure 3, i.e., SERIAL BLOCK USING "HQ";, declares the execution of the outermost block to be assigned to HQ. Nested USING blocks may be assigned to other control points, to the point at which the ultimate action is called for. The SYREAL program expresses this assignment by an AT block, in the sense that the action

```
OBJECT PY CHARACTERISTICS LIKE DEFAULT;
  KIT_ID "PY020";
  PRODUCT_NAME "PY, 2.0";
  TRANSITION FETCH
    CMD "$@RLZ$SCRIPTS:RLZ$FETCH";
  TRANSITION ACTIVATE
    CMD "$@RLZ$SCRIPTS:RLZ$ACTIVATE";
END CHARACTERISTICS PY;
```

Figure 2
SYREAL Program—Object Declaration

```
SERIAL BLOCK USING "HQ";
  ON ERROR SUSPEND;
  SERIAL BLOCK AT "HQ";
    PERFORM FETCH
      OBJECT PY;
  END SERIAL BLOCK AT "HQ";
  CONCURRENT BLOCK USING "HQ";
    SERIAL BLOCK USING "HQ";
      SERIAL BLOCK AT "G1";
        PERFORM COPY
          OBJECT PY
          SERVER "HQ";
      END SERIAL BLOCK AT "G1";
      CONCURRENT BLOCK USING "G1";
        SERIAL BLOCK AT "B1";
          PERFORM COPY
            OBJECT PY
            SERVER "G1";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B1";
        SERIAL BLOCK AT "B2";
          PERFORM COPY
            OBJECT PY
            SERVER "G1";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B2";
      END CONCURRENT BLOCK USING "G1";
    END SERIAL BLOCK USING "HQ";
    SERIAL BLOCK USING "HQ";
      SERIAL BLOCK AT "G2";
        PERFORM COPY
          OBJECT PY
          SERVER "HQ";
      END SERIAL BLOCK AT "G2";
      CONCURRENT BLOCK USING "G2";
        SERIAL BLOCK AT "B3";
          PERFORM COPY
            OBJECT PY
            SERVER "G2";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B3";
        SERIAL BLOCK AT "B4";
          PERFORM COPY
            OBJECT PY
            SERVER "G2";
          PERFORM INSTALL
            OBJECT PY;
        END SERIAL BLOCK AT "B4";
      END CONCURRENT BLOCK USING "G2";
    END SERIAL BLOCK USING "HQ";
  END CONCURRENT BLOCK USING "HQ";
  CONCURRENT TRANSACTION USING "HQ";
    CONCURRENT BLOCK USING "G1";
      SERIAL BLOCK AT "B1";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B1";
      SERIAL BLOCK AT "B2";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B2";
    END CONCURRENT BLOCK USING "G1";
    CONCURRENT BLOCK USING "G2";
      SERIAL BLOCK AT "B3";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B3";
      SERIAL BLOCK AT "B4";
        PERFORM ACTIVATE
          OBJECT PY;
      END SERIAL BLOCK AT "B4";
    END CONCURRENT BLOCK USING "G2";
  END CONCURRENT TRANSACTION USING "HQ";
END SERIAL BLOCK USING "HQ";
```

Figure 3
SYREAL Program—The Executable

is aimed at an individual system. An AT block may contain one or more PERFORM statements, which perform the action called for. The second executable line in Figure 3, i.e., SERIAL BLOCK AT "HQ";, calls for the fetch transition on the object PY. This action results in execution of the command @RLZ$SCRIPTS:RLZ$FETCH on HQ to fetch the distribution kit files from the software library.

A transaction is simply a block that enforces the fault action ON ERROR ROLLBACK. Nested operations must complete successfully or all will roll back. A transaction may be serial or concurrent and may contain nested blocks that are serial or concurrent. It may not contain a nested transaction.

Deployment Processing Control point and target servers are implemented on each OpenVMS system in the network by a single server daemon called the realization server (RLZ). On receipt of the SYREAL program, the first daemon, which is on HQ, converts the program to a binary representation on disk. This data file mirrors the nesting structure of the text file but allows for storage of additional state information.

The daemon then executes the program by sending the binary version of each block that is currently eligible for execution to the block's designated control point. Each control point that receives a binary block repeats this process, until an AT block arrives at its designated control point. The control point then sends to the target system's daemon a request to perform the action. The target daemon creates a process to execute the PERFORM command, captures completion status when the process exits, and returns the status to the control point. If the perform action is successful, the control point sends the next perform request. If the perform action fails, the control point decides whether to send the next perform request, to suspend processing until an operator can intervene, or to initiate a rollback. This decision depends on the fault action in effect.

The RLZ daemon maintains processing state on disk to allow recovery from system failures, loss of network connectivity, and other transient calamities. As block processing completes, block status is rolled up to its containing block, whether local or on a remote control point. The state of the block changes to reflect the block's interpretation of the states of its nested blocks. At each level, the control point decides if, as a result of status reports, one or more additional blocks should be executed. Ultimately, the central control point at HQ will have received the status of all operations. If all the perform actions completed successfully, as determined by the fault actions specified, the deployment completes successfully. Otherwise, the deployment fails. Completion triggers execution of a PCM update script.

PCM Update The overall status of a Project Gabriel realization is an interpretation of the results of many individual operations, some governed by fault actions different from those of the others. Because CGN dynamically generates the block structure of a realization program, the structure has no direct counterpart in the PCM. Therefore, only the results of individual perform actions are of interest for updating the PCM. The update program examines the completion status of each perform action completed on each object on each target system. The program updates the corresponding objects in the PCM based on the results of the last action completed on each object.

Note that since object and transition definitions are specific to a particular SYREAL program, realization servers are not limited to the object classes that Project Gabriel's CGN and PCM update handle. Applications can be written to perform other kinds of operations with new object classes, transitions, etc.

Realization Block Diagram Figure 4 illustrates the complete processing that the RLZ servers carry out in response to the example SYREAL program in the case where no faults occur. Events flow from left to right. The outermost block contains all the events of interest except PCM update, which is implicit in every SYREAL program and carried out automatically by the RLZ server at the root of a deployment operation.

The first action to be executed within the outermost block is fetching PY from the library to staging storage on HQ, under the control of HQ. Subsequently, HQ controls concurrent operations to copy PY from HQ to both G1 and G2. When the copy action is completed on either G1 or G2, HQ transfers the next block to the respective control point to perform the copy and install actions on its two targets. For instance, the concurrent block using G1 executes the copy action to B1 and then the install action on B1, while the same sequence executes on B2. Processing of these concurrent sequences synchronizes on G1 when both complete. At that time, the status of the entire concurrent block using G1 rolls up to HQ, where processing will again synchronize with the concurrent block using G2.

HQ also executes the concurrent transaction. This execution flows similarly to the preceding concurrent block execution except that since no action needs to be taken on G1 or G2 before proceeding to act on B1, B2, B3, and B4, the serial blocks at G1 and G2 are unnecessary.

Fault Handling In the deployment example, the fault action represented by the command ON ERROR SUSPEND governs the steps prior to the transaction. This means that, if an action fails, no dependent action

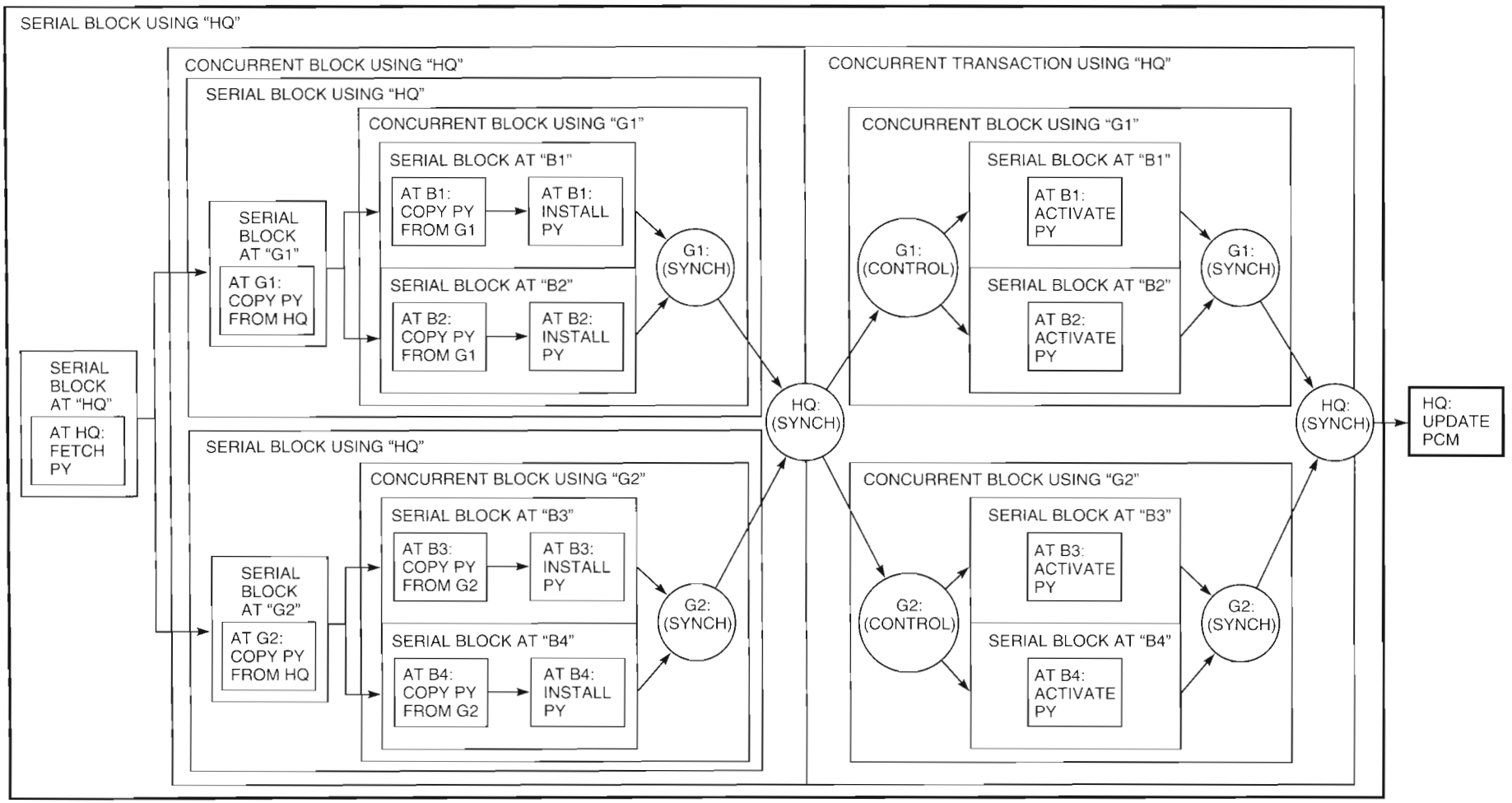


Figure 4
Realization Block Diagram

will be performed. Instead, an event message will be sent up the control tree to HQ. An operator can detect this condition (either as a result of the event message or during a periodic status check), intervene to correct the problem, and restart the action that failed. For example, if the copy action of PY to B1 from G1 fails, the first serial block at B1 will be suspended and the action to install PY on B1 will not be performed. (The install action follows the copy action in a serial block because it is dependent upon successful completion of the copy action.) The blocks in the first part of the deployment, i.e., the serial block at B2 and the concurrent block using G2, continue to execute, however. No processing will go beyond the first HQ synchronization point until the fault is corrected and the serial block at B1 completes. If the problem cannot be corrected, the deployment can be stopped and replanned, perhaps excluding the node that failed.

If one of the actions in the concurrent transaction fails, no additional actions within the transaction will be started and any that completed, including the failed one, will be rolled back. Each transition may have an associated ROLLBACK command. The rollback of an action consists of executing its ROLLBACK command. (This command is not shown in the SYREAL sample.) In this case, the ROLLBACK command deactivates PY. If the transaction has more activations, any that start before the failure is detected are rolled back in the reverse order of execution. The RLZ server effectively runs the transaction in reverse, from the point at which the failure was detected, executing the ROLLBACK command for each action that had completed. To accomplish this, each control point that detects a failure within a transaction or receives a rollback request from one of its subordinate control points initiates a rollback in all the parts of the transaction under its control. At the same time, the control point sends a rollback request to its control point. This process continues until the rollback request reaches the control point that controls the outermost block of the transaction.

A Note about Testing

Consider the challenge of testing a deployment system designed to operate over hundreds or thousands of systems. The PCM and CGN components are centralized, so load testing and boundary testing are relatively straightforward. Executing deployment operations is an inherently distributed process, however, with one RLZ server per host. The team designed the RLZ server to isolate all its data, e.g., network object name, log files, deployment program state data, and command procedures, based on the name given the server process. This design enabled the team to run as many copies of the server on a single system as the system's resources allowed—one VAXstation 4000 system was able to run more than 250 simultaneous servers—and to execute dummy command procedures. Such a design allowed the team to test elaborate simulated deployments and forced it to design the server to deal with a number of unusual resource shortages.

Project Gabriel's performance data indicated that the overhead of the RLZ server was relatively insignificant when compared with that of the actions performed by means of command procedures. This data supported the team's belief that the system would be scalable: A target system that has the resources to support relatively resource-intensive actions like software installations can support one RLZ server to automate the installations.

Conclusions

This paper does not cover topics such as the complex rules regarding the suspension/resumption and restart of operations, lost server time-outs, and interim status updates. Also, the PCM data is considerably more complex than the discussion indicates, as is the asynchronous processing implemented in the RLZ server and the logic of CGN.

A great deal of detail has been omitted in order to focus on the usefulness of a particular collection of abstractions in solving a difficult problem. The entity model and the configuration management model helped to define, partition, and communicate about the problem. The distribution model from the POLYCENTER Software Distribution advanced development work provided essential ideas that the other models did not. These intellectual assets were instrumental in fulfilling the customer's requirements. In "What Good are Models, and What Models are Good?" Fred B. Schneider asserts: "Distributed systems are hard to design because we lack intuition for them."[6] By formulating and analyzing an abstract model, we can develop such intuition, but it is a slow process. It is easy to underestimate both its difficulty and its value.

The model of distributed process control developed for Project Gabriel has proven useful and versatile. It could be profitably applied to the design of a process control service for distributed object technology, such as the Object Management Group's Common Object Request Broker Architecture (CORBA).[7] In such a design, instead of executing a command procedure to perform an action, a process control daemon would invoke a CORBA request on an object. Programs become nested collections of requests with associated state. Improving distributed object and object-oriented database technology should make possible

fuller realization of the PCM and a more powerful CGN. The work accomplished in Project Gabriel just scratched the surface.

Summary

By applying relatively well-developed conceptual models for network and system management, Project Gabriel successfully implemented automated software deployment in a large commercial network. The result is a scalable, distributed system management application that can be used to solve a variety of complex distributed system management problems.

Acknowledgments

The following individuals contributed directly to the design and implementation of the Project Gabriel software: Cecile Beyh, Hugo Diaz, Gill Haik, Paul Kelsey, Michel Lazarescu, John Lundgren, Pat Madden, Mary Maling, Keith Noddle, Laura Spink, Son Voba, David Wihl, and Michael Young.

References

1. *Information Technology—Data Communications—X.25 Packet Layer Protocol for Data Terminal Equipment,* ISO/IEC 8208:1990 (Geneva: International Organization for Standardization/International Electrotechnical Commission, 1990).
2. *Interface between Data Terminal Equipment and Data Circuit-terminating Equipment for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuits—Data Communication Networks: Services and Facilities, Interfaces,* Recommendation X.25-89 (Geneva: International Telecommunications Union, Comité Consultatif Internationale de Télégraphique et Téléphonique [CCITT], 1989).
3. M. Sylor, "Managing DECnet Phase V: The Entity Model," *IEEE Networks* (March 1988): 30–36.
4. *Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs,* MIL-STD-48A (Washington, D.C.: Department of the United States Air Force, June 4, 1985).
5. J. Rumbaugh, et al., *Object-Oriented Modeling and Design* (Englewood Cliffs, N.J.: Prentice-Hall International, 1991): 457.
6. F. Schneider, "What Good are Models and What Models are Good?" *Distributed Systems,* 2d ed., S. Mullender, ed. (New York: ACM Press, 1993): 17–26.
7. *Common Object Request Broker Architecture Specification,* draft 29, revision 1.2 (Framingham, Mass.: Object Management Group, Document No. 93-12-43, December 1993).

Biography

Owen H. Tallman
Currently employed by NetLinks Technology, Inc., of Nashua, New Hampshire, Owen Tallman worked at Digital Equipment Corporation from 1983 through 1994. As a principal software engineer in the Networked Systems Management Engineering group, he led Project Gabriel. He was a management information architect in the Enterprise Management Architecture group and helped develop the POLYCENTER Software Distribution product (formerly known as the Remote System Manager [RSM] product). Owen holds a B.A. in computer science from Regents College in Albany, New York, and is coauthor of two pending patents on RSM technology. He is a member of ACM.

Referees, March 1994 to January 1995

The editors acknowledge and thank the referees who have participated in a peer review of the papers submitted for publication in the *Digital Technical Journal*. The referees' detailed reports have helped ensure that papers published in the *Journal* offer relevant and informative discussions of computer technologies and products. The referees are computer science and engineering professionals from academia and industry, including Digital's consulting engineers.

Alan Abrahams, *Digital*
Brian Allison, *Digital*
Marco Annaratone, *Digital*
Nader Bagherzadeh, *University of California, Irvine*
Kenneth Bates, *Digital*
Edward Benson, *Digital*
Thomas Benson, *Digital*
Dileep Bhandarkar, *Digital*
David L. Black, *Open Software Foundation*
Kenneth M. Brown, *Digital*
Wayne Cardoza, *Digital*
Daniel Cobb, *Digital*
Seth Cohen, *Digital*
William V. Courtright II, *Carnegie Mellon University*
Neil Davies, *Digital*
Scott H. Davis, *Digital*
Wolfgang Deiters, *Fraunhofer Institute*
Hans de Jong, *Digital*
Alexis Delis, *Queensland University of Technology*
Jeremy Dion, *Digital*
Leonard Fehskens, *Digital*
John Forecast, *Digital*
Tryggve Fossum, *Digital*
Derek Frankforth, *Forte Software*
Les Gasser, *University of Southern California*
Jim Gray
Robert Gries, *Digital*
James Grochmal, *Digital*
William Grundmann, *Digital*
Robert Halstead, *Digital*
Charles Hammond, *Digital*
Mark Heinrich, *Stanford University*
Daniel Hirschberg, *University of California, Irvine*
Paul Huntwork, *Digital*
Michael Kantrowitz, *Digital*
Brian Keane, *Digital*
Lawrence Kenah, *Digital*
Jeffrey S. Kuskin, *Stanford University*
William Laing, *Digital*
Edward Lee, *Digital*
Michael Leis, *Digital*
Debra Lelewer, *California State Polytechnic University*
Daniel E. Lenoski, *Silicon Graphics Computer Systems*
Thomas Levergood, *Open Market, Inc.*
Saul Levy, *Rutgers University*
Woody Lichtenstein, *Silicon Graphics, Inc.*
Peter Lucas, *MAYA Design Group, Inc.*
Christopher Marshall, *Digital*
Barry A. Maskas, *Digital*
John McDermott, *Digital*
William Michalson, *Worcester Polytechnic Institute*
J. Eliot B. Moss, *University of Massachusetts*
Rishiyur S. Nikhil, *Digital*
M. Tamer Ozsu, *University of Alberta*
David Patterson, *University of California, Berkeley*
Andrew Payne, *Open Market, Inc.*
Mary Payne, *Digital*
Stephen Root, *Digital*
Robert Rowlands, *Digital*
Howard Rubin, *Hunter College*
Paul Rubinfeld, *Digital*
Kenneth Salem, *University of Waterloo*
Will Sherwood, *Digital*
Allen Simons, *Digital*
Arun Somani, *University of Washington*
Thomas Speer, *Digital*
Lawrence C. Stewart, *Open Market, Inc.*
Jan te Kiefte, *Digital*
David Thiel, *Digital*
Peng Tu, *University of Illinois*
David Wecker, *Digital*
Lih Weng, *Digital*
Robert Willard, *Digital*
Richard Witek, *Digital*
Larry D. Wittie, *State University of New York, Stony Brook*
Bruce L. Worthington, *University of Michigan*

Further Readings

The *Digital Technical Journal* is a refereed, quarterly publication of papers that explore the foundations of Digital's products and technologies. *Journal* content is selected by the Journal Advisory Board, and papers are written by Digital's engineers and engineering partners. Engineers who would like to contribute a paper to the *Journal* should contact the Managing Editor, Jane Blake, at RDVAX::BLAKE or blake@rdvax.enet.dec.com.

Topics covered in previous issues of the *Digital Technical Journal* are as follows:

Database Integration, Alpha Servers & Workstations/ Alpha 21164 CPU
Vol. 7, No. 1, 1995, EY-T135E-TJ

RAID Array Controllers/Workflow Models/PC LAN and System Management Tools
Vol. 6, No. 4, Fall 1994, EY-T118E-TJ

AlphaServer Multiprocessing Systems/DEC OSF/1 Symmetric Multiprocessing/Scientific Computing Optimization for Alpha
Vol. 6, No. 3, Summer 1994, EY-S799E-TJ

Alpha AXP Partners—Cray, Raytheon, Kubota/DECchip 21071/21072 PCI Chip Sets/ DLT2000 Tape Drive
Vol. 6, No. 2, Spring 1994, EY-F947E-TJ

High-performance Networking/OpenVMS AXP System Software/ Alpha AXP PC Hardware
Vol. 6, No. 1, Winter 1994, EY-Q011E-TJ

Software Process and Quality
Vol. 5, No. 4, Fall 1993, EY-P920E-DP

Product Internationalization
Vol. 5, No. 3, Summer 1993, EY-P986E-DP

Multimedia/Application Control
Vol. 5, No. 2, Spring 1993, EY-P963E-DP

DECnet Open Networking
Vol. 5, No. 1, Winter 1993, EY-M770E-DP

Alpha AXP Architecture and Systems
Vol. 4, No. 4, Special Issue 1992, EY-J886E-DP

NVAX-microprocessor VAX Systems
Vol. 4, No. 3, Summer 1992, EY-J884E-DP

Semiconductor Technologies
Vol. 4, No. 2, Spring 1992, EY-L521E-DP

PATHWORKS: PC Integration Software
Vol. 4, No. 1, Winter 1992, EY-J825E-DP

Image Processing, Video Terminals, and Printer Technologies
Vol. 3, No. 4, Fall 1991, EY-H889E-DP

Availability in VAXcluster Systems/Network Performance and Adapters
Vol. 3, No. 3, Summer 1991, EY-H890E-DP

Fiber Distributed Data Interface
Vol. 3, No. 2, Spring 1991, EY-H876E-DP

Transaction Processing, Databases, and Fault-tolerant Systems
Vol. 3, No. 1, Winter 1991, EY-F588E-DP

VAX 9000 Series
Vol. 2, No. 4, Fall 1990, EY-E762E-DP

DECwindows Program
Vol. 2, No. 3, Summer 1990, EY-E756E-DP

VAX 6000 Model 400 System
Vol. 2, No. 2, Spring 1990, EY-C197E-DP

Compound Document Architecture
Vol. 2, No. 1, Winter 1990, EY-C196E-DP

Distributed Systems
Vol. 1, No. 9, June 1989, EY-C179E-DP

Storage Technology
Vol. 1, No. 8, February 1989, EY-C166E-DP

CVAX-based Systems
Vol. 1, No. 7, August 1988, EY-6742E-DP

Software Productivity Tools
Vol. 1, No. 6, February 1988, EY-8259E-DP

VAXcluster Systems
Vol. 1, No. 5, September 1987, EY-8258E-DP

VAX 8800 Family
Vol. 1, No. 4, February 1987, EY-6711E-DP

Networking Products
Vol. 1, No. 3, September 1986, EY-6715E-DP

MicroVAX II System
Vol. 1, No. 2, March 1986, EY-3474E-DP

VAX 8600 Processor
Vol. 1, No. 1, August 1985, EY-3435E-DP

Subscriptions and Back Issues

Subscriptions to the *Digital Technical Journal* are available on a prepaid basis. The subscription rate is $40.00 (non-U.S. $60.00) for four issues and $75.00 (non-U.S. $115.00) for eight issues. Orders should be sent to Cathy Phillips, Digital Equipment Corporation, 30 Porter Road LJO2/D10, Littleton, Massachusetts 01460, U.S.A., Telephone: (508) 486-2538, Fax: (508) 486-2444. Inquiries can be sent electronically to dtj@digital.com. Subscriptions must be paid in U.S. dollars, and checks should be made payable to Digital Equipment Corporation.

Single copies and past issues of the *Digital Technical Journal* are available for $16.00 each by calling DECdirect at 1-800-DIGITAL (1-800-344-4825). Recent back issues of the *Journal* are available on the Internet at http://www.digital.com/info/DTJ/home.html.

Digital Research Laboratory Reports

Reports published by Digital's research laboratories can be accessed on the Internet through the World Wide Web or FTP. For access information on the electronic or hard-copy versions of the reports, see http://www.research.digital.com/home.html.

Technical Papers by Digital Authors

N. Arora, R. Rios, and C. Huang, "Modeling the Polysilicon Depletion Effect and Its Impact on Submicron CMOS Circuit Performance," *IEEE Transactions on Electron Devices* (May 1995).

D. Bhavsar and R. Fromm, "Testability Features and Testability Access of the Alpha 21164 Microprocessor," *Proceedings of the IEEE 1995 Custom Integrated Circuits Conference* (May 1995).

W. Bowhill et al., "A 300MHz 64b Quad-Issue CMOS RISC Microprocessor," *1995 IEEE International Solid-State Circuits Conference* (February 1995).

J. Chen, "Clocking PLL Solutions for High Speed Computers," *1995 International Symposium on VLSI Technology, Systems, and Applications* (June 1995).

T. Dalton, "Plasma Diagnostics: Monitoring and Control of Plasma Tools," *American Vacuum Society New England Chapter 1995 Annual Symposium* (June 1995).

M. Davidson and N. Sullivan, "Monte Carlo Simulation For CD SEM Algorithm Development," *Proceedings of the Society of Photo-Optical Instrumentation Engineers (SPIE)—Integrated Circuit Metrology, Inspection, and Process Control IX* (February 1995).

M. Elbert and R. Howe, "Stress Testing Profiles—Which Should I Use," *Thirty-third Annual Spring IEEE Relativity Symposium* (April 1995).

T. Hongsmatip and B. Twombly, "Dynamic Mechanical Analysis of Silver/Glass Die Attach Material," *Forty-fifty Electronic Components and Technology Conference* (May 1995).

C. Huang, N. Arora, N. Khalil, B. Zetterlund, and L. Bair, "Effects of Source/Drain Implants on Short-Channel MOSFET I-V and C-V Characteristics," *IEEE Transactions on Electron Devices* (July 1995).

J. Kitchin, "Statistical Electromigration Budgeting for Reliable Design and Verification in a 300-MHz Microprocessor," *1995 Symposium on VLSI Circuits Digest of Technical Papers* (June 1995).

B. Mirman, "Translation of Stress States into Reliability Terms for Single Chip Ceramic Packages," *Transactions of The American Society of Mechanical Engineers (ASME)—Journal of Electronic Packaging* (December 1994).

A. Philipossian and H. Soleimani, "Determining the Wafer Temperature in Atmospheric Thermal Silicon Oxidation Reactors," *Journal of the Electrochemical Society* (May 1995).

P. Rubinfeld, J. Edmondson, R. Preston, and V. Rajagopalan "Superscaler Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro* (April 1995).

N. Sullivan and S. Arsenault, "SEM Review of Unpatterned Particle Monitor Wafers," *Proceedings of the Society of Photo-Optical Instrumentation Engineers (SPIE)—Integrated Circuit Metrology, Inspection, and Process Control IX* (February 1995).

M. Tsuk and R. Evans, "Modeling and Measurement of a High-Performance Computer Power Distribution System," *IEEE Transactions on Components, Packaging, and Manufacturing Technology* (November 1994).

E. Valcarce and G. Hogland, "The ESSENSE of Intrusion Detection: A Knowledge-Based Approach to Security Monitoring and Control," *Seventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE)* (June 1994).

A. Villani and W. Clark, "Ceramic Strength and Weibull Statistics Variation as a Function of Sample Size," *Proceedings of the International Intersociety Electronic Packaging Conference (Interpak '95)* (March 1995).

R. Walsh and C. Ozveren, "The Gigaswitch Control Processor," *IEEE Network* (January 1995).

A. Westerheim, A. Labun, I. Dubash, J. Arnold, H. Swain, and V. Yu-Wang, "Substrate Bias Effects in High-Aspect-Ratio SiO2 Contact Etching Using an ICP Reactor," *Journal of Vacuum Science and Technology A—Vacuum, Surfaces, and Films* (June 1995).

Digital Press

Digital Press, the authorized publisher for Digital Equipment Corporation, is an imprint of Butterworth-Heinemann, a major international publisher of professional books and a member of the Reed Elsevier group. The following are descriptions of computing titles available from Digital Press.

THE SQL SERVER HANDBOOK—A Guide to Microsoft Database Computing
Ken England and Nigel Stanley, October 1995, paperback, 450 pages, ISBN 1-55558-152-8 ($34.95), EY-T818E-DP.

Microsoft SQL Server for Windows NT is fast becoming the database server of choice for the Windows NT operating system. The latest release of Microsoft SQL Server, Version 6.0, is a sophisticated database server with a wealth of new capabilities including powerful graphical administration of distributed servers, data replication across the network, and many new performance tuning, administration, and data integrity options. SQL Server 6.0 will have a significant impact on the database industry.

The SQL Server Handbook—A Guide to Microsoft Database Computing is essential for anyone involved in the procurement, training, design, administration, implementation, and tuning of SQL Server 6.0 databases. Drawing on the authors' significant practical experience with relational database management systems, this book covers all the major topics necessary to gain a good understanding of the SQL Server, including the new features in SQL Server 6.0. The book also provides information on many other products in the Microsoft database family, such as the Microsoft Access Upsizing Tool, Microsoft ODBC, and the Jet database engine.

Database designers, administrators, programmers, and newcomers to Microsoft SQL Server will find this book an indispensable reference for understanding and utilizing the product. Database professionals studying for Microsoft Certified Professional qualifications will also find this book essential reading.

Ken England is Chief Executive of Database Technologies Limited, a company specializing in database consulting, product evaluation, and training. Nigel Stanley, formerly at Microsoft as European Product Manager, responsible for the Microsoft client server products, is now technical director for ICS Solutions Ltd., a Microsoft solution provider.

ADVANCED ETHERNET/802.3 MANAGEMENT AND PERFORMANCE, Second Edition
Bill Hancock, October 1995, paperback, 400 pages, ISBN 1-55558-144-7 ($34.95), EY-T140E-DP.

Advanced Ethernet/802.3 Management and Performance, Second Edition was designed for users of the Ethernet/802.3 LAN–environment hardware and software to answer the myriad questions that come up after a network is installed. The book addresses questions such as, when do you use bridges and routers to isolate traffic? what are switching bridges and why are they necessary? what are the rules for unshielded twisted-pair networks? how do you know when the performance of the network is suffering, and how do you collect data to prove it? what is "heartbeat," and how is it set? The book also contains information on many other topics essential to the day-to-day management and control of the LAN. This second edition includes information on the new Fast Ethernet (100BASE-T) standard, the new 802.3 fiber standards, the use of switching bridges to improve performance through traffic isolation, and how to clearly identify proper settings of "heartbeat."

Dr. Bill Hancock is a well-known computer and network consultant, designer, and engineer. He has designed and reengineered networks for many of the Fortune 1000 as well as many international companies and governments.

ALPHA AXP ARCHITECTURE REFERENCE MANUAL, Second Edition
Richard Sites and Richard Witek, September 1995, paperback, 864 pages, ISBN 1-55558-145-5 ($49.95), EY-T132E-DP.

Written by the co-designers of the Alpha architecture, the *Alpha AXP Architecture Reference Manual, Second Edition* is a major revision of the first edition. This book includes the original material plus significant new information and changes necessitated by the evolution of the Alpha architecture since the writing of the first edition. The second edition discusses the Windows NT PALcode architecture, 128-bit IEEE floating-point support, and bi-endian support, and contains revised PCC information and console interface section. The significant technical changes include the clarification of Mxx FPCT operand and trap disable flags and of system architecture and programming implications, and the addition of CVTST, WMB, and EXCB instructions.

INFORMATION TECHNOLOGY STANDARDS: The Quest for the Common Byte
Martin Libicki, August 1995, hardcover, 432 pages (est.), ISBN 1-55558-131-5 ($59.95), EY-S422E-DP.

This book examines information technology standards and discusses what they are, what they do, how they originate, and how they evolve. Standards are important in improving system interoperability and thereby increasing economic productivity, but they are unlikely to achieve their full potential due to a variety of factors. Chief among these factors is the politics of the standard process itself. Libicki points out that the government is probably not the best source for designing and promoting standards. He breaks down many complex technical issues and presents them in a fashion that technical people can enjoy and policy makers can understand.

Martin Libicki is a Senior Research Fellow at the National Defense University in Washington, D.C.

SOFTWARE IMPLEMENTATION TECHNIQUES: OpenVMS, UNIX, OS/2 and Windows NT
Don Merusi, September 1995, ISBN 1-55558-134-8, paperback, 608 pages ($49.95), EY-T131E-DP.

Software Implementation Techniques: OpenVMS, UNIX, OS/2 and Windows NT is a comparison of four operating system platforms. The book provides software designers with an introduction on how to migrate comparable program functionality between the different platforms. The book is designed to facilitate determining what is required to implement a specific operating system function. The topics covered include process and thread scheduling, synchronization and concurrency primitives, file management, memory management, performance, networking facilities, and user interfaces.

Don Merusi has been a senior computer systems support specialist for 22 years. Currently, he is responsible for administering large-scale PC LANs using PATHWORKS,

Windows for Workgroups, and Windows NT. Mr. Merusi is also an associate adjunct professor at the Hartford Graduate Center and teaches courses on operating systems.

WRITING DEVICE DRIVERS: Tutorial and Reference

Tim Burke, Mark A. Parenti, and Al Wojtas, April 1995, paperback, 1,140 pages, ISBN 1-55558-141-2 ($69.95), EY-S796E-DP.

Writing Device Drivers: Tutorial and Reference discusses how to write device drivers for computer systems running the Digital UNIX operating system (formerly called the DEC OSF/1 operating system). By following the task-oriented approach, the reader will acquire the skills necessary to write OSF-based device drivers. The book provides information on designing drivers, OSF-based data structures, and OSF-based kernel interfaces, and contains source code listings for the driver examples and a glossary. Mastery of the concepts and examples presented in the book provides a fundamental background for writing a variety of device drivers, including disk and tape controllers and more specialized drivers such as array processors.

Tim Burke is a principal software engineer, Mark A. Parenti is a consulting software engineer, and Al Wojtas is a principal software technical writer, all at Digital Equipment Corporation.

ALPHA IMPLEMENTATION AND ARCHITECTURE

Dileep Bhandarkar, October 1995, paperback, 400 pages, ISBN 1-55558-130-7 ($39.95), EY-T141E-DP.

Alpha Implementation and Architecture provides a comprehensive description of all major aspects of Alpha systems. The book includes an overview of the history of RISC development in the computer industry and at Digital, the Alpha architecture, all the major processor chips, and system implementations. The book covers RISC concept and design styles, and provides an overview of other RISC architectures and descriptions of the new SPARC, MIPS, PowerPC, and PA-RISC microprocessors introduced in 1995. The book also discusses operating system porting issues, compiler techniques, and binary translation. Practicing computer engineers and graduate students in computer architecture will find this reference book invaluable because it describes the trade-offs and design philosophy that lead to the development of the Alpha architecture and its implementation.

Dileep Bhandarkar was a senior consulting engineer at Digital Equipment Corporation. He led the technical direction and product strategy of Alpha Personal Systems, Alpha and VAX Servers, and High Performance Computing. He was the architecture manager for MicroVAX, chief architect for VAX vector processing, and co-architect of the PRISM RISC architecture on which Alpha is based. He currently works for Intel Corporation.

OPENVMS SYSTEM MANAGEMENT GUIDE

Lawrence Baldwin, October 1995, paperback, 416 pages (includes diskette), ISBN 1-55558-143-9 ($44.95), EY-T119E-DP.

This book provides a comprehensive description of OpenVMS system management tasks and is geared toward showing systems managers how to manage smarter by automating wherever possible and being proactive rather than reactive. Basic management procedures are not only documented but also prioritized as to what should be done and why. Specific procedures are provided to automate or simplify system management tasks.

Lawrence Baldwin, an independent consultant, is the President of System Management Technologies.

DESIGNING AND DEVELOPING ELECTRONIC PERFORMANCE SUPPORT SYSTEMS

Lesley A. Brown, October 1995, paperback, 250 pages, ISBN 1-55558-139-0 ($29.95), EY-T126E-DP.

Designing and Developing Electronic Performance Support Systems describes the EPSS concept and provides a systematic process for creating these systems. An EPSS is a software context that integrates the support needed to perform a job task—information, software, and expert advice—with the actual job task or tasks. EPSSs provide this support at the appropriate time and in the most appropriate format. As corporations cut their training budgets and realize the relevance of on-the-job support, there is growing acceptance of the EPSS as an alternative to classroom-based training.

ED4 (EPSS Define, Design, Develop, and Deliver), a systematic approach to creating EPSS, is based on instructional systems methodology, and was used at Digital Equipment Corporation to create an EPSS "workbench" for training consultants. This book describes ED4 and the process that the instructional designers and software engineers used to create the Learning Services Workbench. Interviews with Digital's EPSS designers and developers showed that EPSSs created using a systematic approach resulted in a creative, robust, and job-relevant software product.

Lesley Brown is an instructional design contractor for the Information Design and Consulting group at Digital Equipment Corporation.

ADVANCED WORDPERFECT USING MACRO POWER, A Guide for VMS and DOS Users

Sharilyn Due, September 1995, paperback, 400 pages (includes a DOS version 6.0 diskette), ISBN 1-55558-147-1 ($36.95), EY-T817E-DP.

Advanced WordPerfect Using Macro Power concentrates on the use of macros for users of any version of WordPerfect in the OpenVMS and DOS environments. The book helps the WordPerfect user save time and become more productive through the use of macros. It covers a series of advanced topics and then provides macro examples to automate the task. Explanations, screen captures, and keystroke captures give the reader an easy-to-follow, step-by-step procedure. After providing an example macro for a task, the author offers other possibilities for reader-created macros. The book covers a diverse range of applications and includes a thorough treatment of how to create, edit, and debug macros.

Recent Digital U.S. Patents

The following patents were recently issued to Digital Equipment Corporation. Titles and names supplied to us by the U.S. Patent and Trademark Office are reproduced exactly as they appear on the original published patent.

D335,501	R. Faranda	Printer Enclosure
D337,760	G. Schneider	Combined Media Cartridge Loader and Associated Magazine
D341,826	M. J. Falkner, R. Hanson, K. Korellis, and C. Danemayer	Computer Enclosure
5,208,518	H. Grapenthin and H. Haug	DC-DC Boost Converter for Spindle Motor Control
5,210,741	R. Grochmal	Low Cost ISDN Switch
5,210,829	H. Bitner	Adjustable Threshold for Buffer Management
5,210,837	C. Wiecek	Methods and Apparatus for Transforming Machine Language Program Control into High-level Language Constructs by Manipulating Graphical Program Representations
5,212,788	D. Lomet, P. Bernstein, J. Johnson, and K. Wilner	System and Method for Consistent Timestamping in Distributed Computer Databases
5,214,553	K. Kan, G. Saliba, and R. Nute	Magnetic Contact Recording Head for Operation with Tapes of Varying Thicknesses
5,216,556	M. Steinberg and G. Saliba	Method for Optimized Tape Tension Adjustment for a Tape Drive
5,220,604	M. Gasser, A. Goldstein, and C. Kaufman	Method for Performing Group Exclusion in Hierarchical Group Structures
5,222,197	H. Teng, K. Chen, M. Wilson, M. Verdeven, and G. Abbruzzese	Rule Invocation Mechanism for Inductive Learning Engine
5,224,106	L. Weng	Multi-level Error Correction System
5,224,163	M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson	Method for Delegating Authorization from One Entity to Another through the Use of Session Encryption Keys
5,225,833	E. Fisher and P. Gilbert	Character Encoding
5,226,150	D. Sanders, M. Callander, and L. Chao	Apparatus for Suppressing an Error Report from an Address for Which an Error Has Already Been Reported
5,226,966	K. Ishibashi, H. Sato, and M. Mallary	Apparatus for Providing Uniaxial Anistrophy in a Magnetic Recording Disk (This case was combined with 90-0812. Japan claims partial priority. Mallary added.)
5,227,778	G. Visser and J. Vacon	Service Name to Network Address Translation in Communications Network
5,228,083	P. Lozowick and S. Ben-Michael	Cryptographic Processing in a Communication Network, Using a Single Cryptographic Engine
5,229,575	D. Waller, L. Colella, and R. Pacheco	Thermode Structure Having an Elongated, Thermally Stable Blade
5,230,044	X. Cao, A. Mohammad, N. Quaynor, and F. Colon-Osorio	Arbitration Apparatus for Shared Bus
5,231,552	G. Schneider and K. Paulat	Magazine and Receiver for Media Cartridge Loader
5,233,616	M. Callander	Write-back Cache with ECC Protection
5,235,211	W. Hamburgen	Semiconductor Package Having Wraparound Metallization

5,235,644	A. Gupta, J. Tardo, C. Kaufman, B. Lampson, W. Hawe, M. Kempf, M. Gasser, and B. J. Herbison	Probabilistic Cryptographic Processing Method (This case was combined with PD90-0295.)
5,237,662	K. Green, S. Jenness, and T. Carruthers	System and Method with a Procedure Oriented Input/Output Mechanism
5,237,673	D. A. Orbits, K. D. Abramson, and H. B. Butts	Memory Management Method for Coupled Memory Multiprocessor Systems
5,239,423	S. Sadowski	Method and Apparatus for Converting Analog Signals into Digital Signals
5,239,493	S. Sherman	Method and Apparatus for Interpreting and Organizing Timing Specification Information
5,239,630	X. Cao, M. Abidi, N. Quaynor, R. Lary, and F. Colon-Osorio	Shared Bus Arbitration Apparatus Having a Deaf Node
5,240,740	K. Frey and M. Mallary	Method of Making a Thin Film Head with Minimized Secondary Pulses
5,241,621	R. Smart	Management Issue Recognition and Resolution Knowledge Processor
5,243,592	R. Perlman and G. Harvey	Method and Apparatus for Distance Vector Routing on Datagram Point-to-Point Links
5,246,294	C. Pan	Flow-regulating Hydrodynamic Bearing
5,247,510	N. Lee, Q. Lam, and P. Van Roekens	Increasing Storage Density of Optical Data Media by Detecting a Selected Portion of a Light Spot Image Corresponding to a Single Domain
5,247,522	F. Reiff	Fault Tolerant Bus
5,247,524	R. Callon	Method for Generating a Checksum
5,249,090	W. Fehse	Disk Storage with Device for Fixing the Disk Pack on Its Hub Such That It Can Be Removed
5,251,205	R. Callon, E. Rosen, R. Perlman, and J. Harper	Multiple Protocol Routing
5,251,310	N. Warchol, D. Smelser, and G. Lidington	Method and Apparatus for Exchanging Blocks of Information between a Cache Memory and a Main Memory
5,253,203	H. Partovi and M. Case	Subarray Architecture with Partial Address Translation
5,255,381	T. P. Fissette, K. Chinnaswamy, H. A. Collins, M. B. Evans, M. A. Gagliardo, J. J. Lunch, and J. E. Tessari	Mode Switching for a Memory System with a Diagnostic Scan
5,261,059	W. Hedberg, M. Halvorson, D. Ellsworth, R. Lewis, P. Brooks, and G. Mendelsohn	Crossbar Interface for Data Communication Network
5,261,066	N. P. Juoppi and R. A. Eustace	Data Processing System and Method with Small Fully Associative Cache and Prefetch Buffers
5,261,097	P. D. Saxon	Computer System and Method for Executing Command Scripts Using Multiple Synchronized Threads
5,262,344	K. R. Mistry	N-Channel Clamp for ESD Protection in Self-aligned Silicided CMOS Process
5,263,127	W. Barabash, S. A. Kirk, W. S. Yerazunis, and K. A. Gilbert	Method for Fast Rule Execution of Expert Systems
5,263,144	R. Ramanujan, J. DeRosa, J. H. Zurawski	Method and Apparatus for Sharing Data between Processors in a Computer System
5,263,160	J. A. Porter, D. E. Matthews, and D. E. Haugh	Augmented Doubly Linked List Search and Management Method for a System Having Data Stored in a List of Data Elements in Memory
5,265,104	L. Weng	Data Storage System including Redundant Storage Devices
5,265,229	B. K. Sareen	Single Load, Multiple Issue Queue with Error Recovery Capability
5,267,349	W. Barabash, S. A. Kirk, and W. S. Yerazunis	Fast Determination of Subtype Relationship in a Single Inheritance Type Hierarchy
5,272,394	J. Kirk and J. Barrett	Wide Bandwidth Peak Follower Circuitry
5,274,783	K. A. House, J. Kirk, and L. Narhi	SCSI Interface Employing Bus Extender and Auxiliary Bus
5,276,852	D. Sanders and M. Callander	Method and Apparatus for Controlling a Processor Bus Used by Multiple Processor Components during Writeback Cache Transactions
5,276,863	G. K. Heider	Computer System Console

5,276,868	N. T. Poole	Method and Apparatus for Pointer Compression in Structured Databases
5,276,877	K. S. Friedrich and A. R. Bousquet	Dynamic Computer System Performance Modeling Interface
5,277,756	J. B. Dion	Post Fabrication Processing of Semiconductor Chips
5,278,727	S. M. Westbrook and G. Howell	High Density Electrical Interconnection Device and Method Therefor
5,278,829	K. J. Dunlap	Reduced Broadcast Algorithm for Address Resolution Protocol
5,280,610	R. L. Travis, A. P. Wilson, N. F. Jacobson, M. J. Renzullo, and A. N. Ewald	Methods and Apparatus for Implementing Data Bases to Provide Object-oriented Invocation of Applications
5,283,571	H. Yang, G. P. Koning, W. R. Hawe, and J. D. Hutchison	Testing a Communications Network for Duplicate Station Addresses
5,283,875	W. Thorsted, R. Lary, K. Gibson, and J. Jackson	Methods and Apparatus for Optimizing Prefetch Caching by Reverse Ordering of Logical Blocks
5,285,323	R. C. Hetherington, F. X. McKeen, J. D. Marci, T. Fossum, and J. S. Emer	Integrated Circuit Chip Having Primary and Secondary Random Access Memories for a Hierarchical Cache
5,285,347	P. C. Wade and L. Fox	Hybrid Cooling System for Electronic Components
5,289,409	R. Reinschmidt	Bipolar Transistor Memory Cell and Method
5,289,567	J. W. Roth	Computer Apparatus and Method for Finite Element Identification in Interactive Modeling
5,291,491	J. D. Hutchison and H. S. Yang	Avoidance of False Re-initialization of a Computer Network
5,291,497	E. G. Ulrich, K. P. Lentz, and M. M. Gustin	Method for Testing, Debugging, and Comparing Computer Programs Using Concurrent Simulation of Program Paths
5,301,163	S. C. Sullivan and R. M. Reinschmidt	Memory Selection/Deselection Circuitry Having a Wordline Discharge Circuit
5,303,265	P. T. McLean	Frequency Independent Encoding Technique and Apparatus for Digital Communications
5,303,362	D. A. Orbits, K. D. Abramson, and H. B. Butts	Coupled Memory Multiprocessor Computer System including Cache Coherency Management Protocols
5,304,845	S. E. Lindquist and D. A. Bailey	Apparatus for an Air Impingement Heat Sink Using Secondary Flow Generators
5,305,161	M. Giovanetti, K. Veseskis, B. Rub, and F. Zayas	Adaptive Track Seeking for Disk Drives
5,305,305	F. Dolan and J. A. Harper	Message Swiching Network Monitoring
5,305,306	H. S. Yang, W. R. Hawe, and B. S. Spinney	Station-to-Station Full Duplex Communication in a Token Ring Local Area Network
5,307,468	T. Schlage	Data Processing System and Process for Controlling the Latter As Well As a CPU Board
5,307,479	E. Ulrich and K. Lentz	Method for Multi-Domain and Multi-Dimensional Concurrent Simulation Using a Digital Computer
5,307,504	S. G. Robinson and R. L. Sites	System and Method for Preserving Instruction Granularity When Translating Program Code from a Computer Having a First Architecture to a Computer Having a Second Reduced Architecture during the Occurrence of Interrupts Due to Asynchronous Events
5,307,506	R. P. Colwell, J. O'Donnell, D. B. Parworth, and P. K. Rodman	High Bandwidth Multiple Computer Bus Apparatus
5,309,437	R. J. Perlman and G. P. Koning	Bridge-like Internet Protocol Router
5,313,581	D. Giokas and A. Leskowitz	System and Method for Communication between Windowing Environments
5,313,623	K. Chinnaswamy, H. A. Collins, M. B. Evans, M. A. Gagliardo, J. J. Lynch, J. E. Tessari, and T. P. Fissette	Method and Apparatus for Performing Diagnosis Scanning of a Memory Unit Regardless of the State of the System Clock and without Affecting the Store Data
5,314,596	H. B. Shukovsky, S. Batra, and M. L. Mallary	Magnetic for Fabricating Magnetic Film Recording Head for Use with a Magnetic Recording Media
5,315,657	M. Abadi, A. C. Goldstein, and B. W. Lampson	Compound Principals in Access Control Lists
5,317,718	N. Jouppi	Data Processing System and Method with Prefetch Buffers
5,317,719	B. A. Rozmovits	Data Format for Packets of Information

Call for Authors from Digital Press

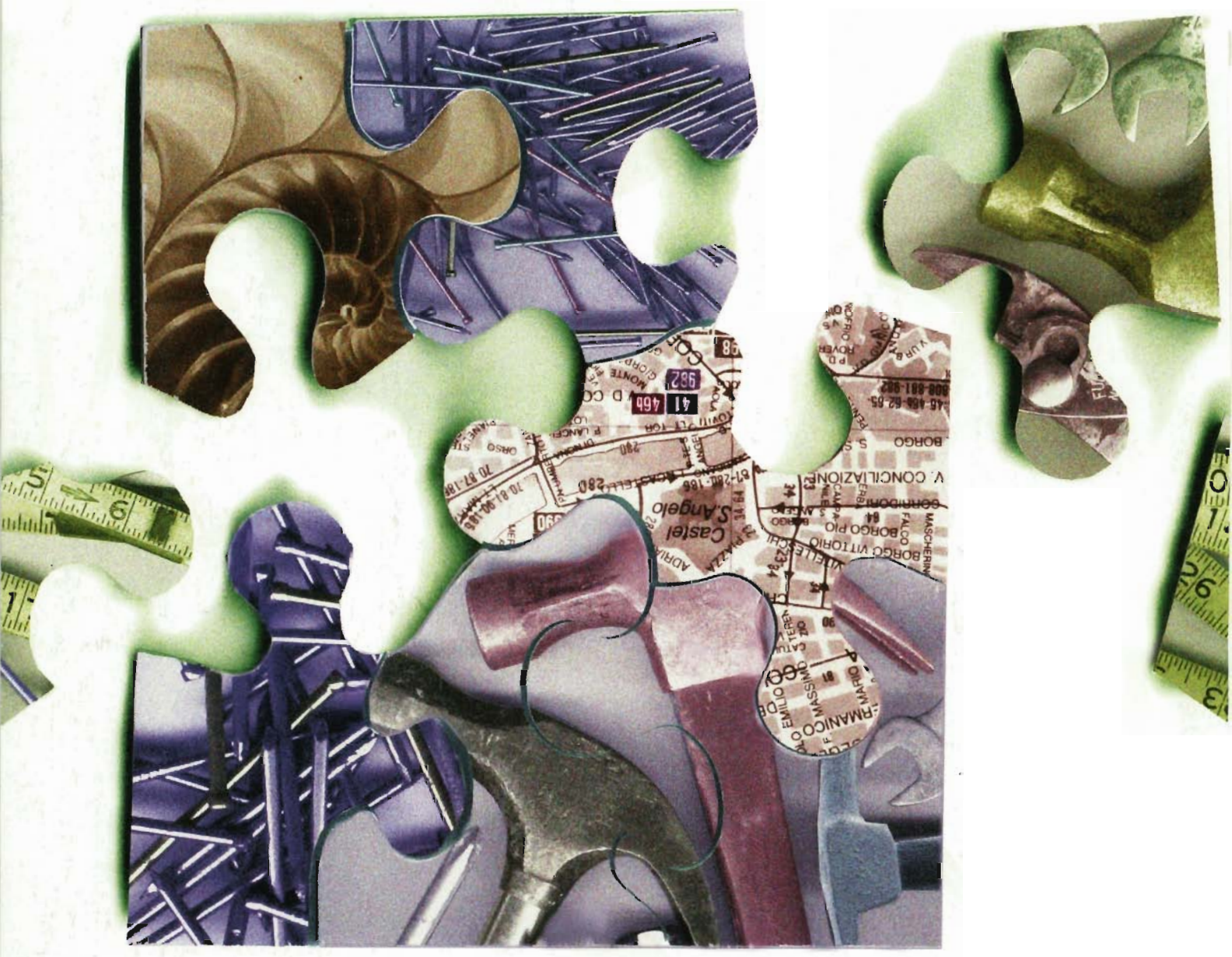
Digital Press is an imprint of Butterworth-Heinemann, a major international publisher of professional books and a member of the Reed Elsevier group. Digital Press is the authorized publisher for Digital Equipment Corporation: The two companies are working in partnership to identify and publish new books under the Digital Press imprint and create opportunities for authors to publish their work.

Digital Press is committed to publishing high-quality books on a wide variety of subjects. We would like to hear from you if you are writing or thinking about writing a book.

Contact: Mike Cash, Digital Press Manager, or
Liz McCarthy, Assistant Editor

DIGITAL PRESS
313 Washington Street
Newton, MA 02158-1626
U.S.A.
Tel: (617) 928-2649, Fax: (617) 928-2640
E-mail: Mike.Cash@BHein.rel.co.uk or
LizMc@world.std.com

digital™



ISSN 0898-901X